Copy No. 9

DEFENCE R&D DÉFENSE

IMPLEMENTATION OF A PSEUDO-COMPRESSIBILITY NAVIER-STOKES SOLVER IN TRANSOM

D. Hally

DEFENCE RESEARCH ESTABLISHMENT ATLANTIC

Technical Memorandum

DREA TM 98/239

December 1998

National Defence Défense nationale

Canada

102-02763/

National Defence
Research and Development Branch

Défense nationale
Bureau de recherche et développement

IMPLEMENTATION OF A PSEUDO-COMPRESSIBILITY NAVIER-STOKES SOLVER IN TRANSOM

David Hally

December 1998

Approved by: J.L. Kennedy:
Head / Warship Signatures and Safety Section

TECHNICAL MEMORANDUM 98/239

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

Abstract

TRANSOM is a multi-block, multi-method Reynolds-averaged Navier Stokes solver being developed at DREA to address problems associated with the flow around ships and submarines. It is multi-block because the flow is divided into several distinct regions. It is multi-method because a different solution method may be used on each of the flow regions. At present two different methods of solution can be chosen: a finite volume solver based on the pseudo-compressibility method; and a finite element solver which uses the penalty function method to determine the pressure.

TRANSOM is written in C++ following principles of Object Oriented Programming. This document describes the design of the finite volume pseudo-compressibility flow solver in TRANSOM with emphasis on the class hierarchies used to represent structured blocks, structured matrix solvers, and the pseudo-compressibility solver itself. Two companion reports describe the overall design of TRANSOM and the design of the finite element solver.

Résumé

TRANSOM est un logiciel de calcul multi-blocs, multi-méthodes Navier-Stokes pondéré Reynolds, développé au CRDA pour résoudre des problèmes de l'écoulement autour des navires et des sous-marins. Il est multi-blocs parce que l'écoulement est divisé en plusieurs régions distinctes. Il est multi-méthodes parce qu'une méthode de résolution différente peut être utilisée avec chacune des régions d'écoulement. À l'heure actuelle, on peut choisir entre deux méthodes: un logiciel à volumes finis basé sur la méthode de la pseudo-compressibilité, et un logiciel à élements finis qui fait appel à la fonction de pénalité pour déterminer la pression.

TRANSOM est écrit en C++ suivant les principes de la programmation orientée objet. Ce document décrit la conception du logiciel d'écoulemnt en pseudo-compressibilité à volume fini de TRANSOM en mettant l'accent sur les hiérarchies de classe utilisées pour représenter les blocs structurés, les logiciels de matrices structurées et le logiciel de pseudo-compressibilité lui-même. Deux rapports parallèles décrivent la conception générale d'ensemble de TRANSOM et la conception du logiciel à éléments finis.

DREA Technical Memorandum TM/98/239

Implementation of a Pseudo-Compressibility Navier-Stokes Solver in TRANSOM

by

David Hally

Executive Summary

Background

The computer program TRANSOM is intended to solve a number of ship-related flow problems of importance to DND: for example, flows around submarines, vortex generation from propellers and control surfaces, bilge vortex generation, and prediction of roll damping coefficients. In particular it will enhance our ability to predict the flow around a ship hull and into the propellers. Knowledge of this flow is crucial for the design of efficient quiet propellers.

TRANSOM is a multi-block, multi-method Reynolds-averaged Navier Stokes solver; multi-block because the flow is divided into several distinct regions called blocks; multi-method because a different solution method may be used on each of the blocks. Unlike the methods currently in use at DREA, TRANSOM solves the full Navier-Stokes equations in conjunction with a turbulence model; hence, when completed, it will be able to solve the flow around complex geometries such as a fully appended ship hull or submarine. The overall design of TRANSOM is described in a companion document.

The current version of TRANSOM includes two different solution methods which can be used on the blocks. One uses the finite element method and is based on the program MEF developed at Université Laval; it is suitable for use on unstructured blocks. MEF was modified at DREA to incorporate variants of the k-ε turbulence model; however, the version that is currently incorporated in TRANSOM is restricted to laminar flow. Details of the implementation of this solver in TRANSOM will be documented separately.

Principal Results

The second solver is described in detail in this memorandum. It is a pseudo-compressibility method based on the work of Rogers and Kwak and developed for DREA under contract by the University of Toronto; it is suitable for use on structured blocks. Turbulence can be modelled using a variant of the k-ε model or using the Baldwin-Lomax model. Currently only two-dimensional flow can be modelled.

TRANSOM is written in C++ following principles of Object Oriented Programming. This document describes the implementation of the pseudo-

compressibility flow solver in TRANSOM and, in particular, the class hierarchies used to represent unstructured blocks, elements, and finite element solvers.

Significance of Results

When TRANSOM is fully developed it will provide accurate numerical predictions of the flow into the propeller plane. It will also be applicable to many other flow problems related to ships: for example, flow around submarines, vortex generation from propellers and control surfaces, bilge vortex generation, prediction of roll damping coefficients, and prediction of the lift and drag characteristics of lifting surfaces such as submarine control planes, rudders, and fin stabilizers.

Future Work

TRANSOM is still under development; the current version will only solve two-dimensional flows and requires upgrading in many areas, in particular the ways in which different solution methods interact at their common block boundaries. Further development will be done at DREA and under contract.

Table of Contents

Abstract ii
Résumé ii
Executive Summary iii
Table of Contents v
List of Figures vii
1 Introduction 1
2 An Overview of the Pseudo-Compressibility Method 4
3 An Overview of the Class Structure Used for SPC Solvers 6
4 Structured Blocks 9
4.1 Structured Node Organizers 9
4.1.1 The BaseS2dNodeOrg Class 10
4.1.2 The S2dNodeOrg Class 12
4.1.3 The S1dNodeOrg Class 15
4.1.4 The ConnS2dNodeOrg Class 16
4.2 The S2dBlock Class 17
4.3 The ConnS2dBlock Class 18
4.4 Blocks with O Topology 20
4.5 Blocks with C Topology 22
5 Classes to Represent and Solve the Linear System 25
5.1 The SPCMtx Class 26
5.2 The PreSPCMtx Class 28
5.3 The SPCMtxSolver Class 29
6 Boundary Conditions 31
6.1 The BdyCond Class 31
6.2 The SPCBdyCond Class 32
7 The Pressure-Velocity Solver 37
7.1 The SPC2dPVSweep Class 37
7.2 The SPC2dPVSolver Class 41
8 Turbulence Solvers 43
8.1 Turbulence Models 43
8.1.1 Turbulence Regions 43
8.1.1.1 The TurbRegion Class 43

8.1.1.2 The STurbRegion Class 45
8.1.2 The WallDistIOVar Class 46
8.1.3 The TurbModel Class 47
8.1.4 The S2dTransTurbModel Class 47
8.2 The Baldwin-Lomax Solver 49
8.2.1 The BLTurbModel Class 49
8.2.2 The S2dBLSweep Class 51
8.2.3 The S2dBLSolver Class 52
8.3 The Spalart-Allmaras Solver 53
8.3.1 The SAViscIOVar Class 54
8.3.2 The SATurbModel Class 55
8.3.3 The S2dSASweep Class 56
8.3.4 The S2dSASolver Class 59
8.4 The Chen-Patel Solver 60
8.4.1 The CPTurbModel Class 61
8.4.2 The S2dCPSweep and S2dCPSolver Classes 63
8.5 Turbulence Solver Sweep Connections 63
9 The Full Pseudo-Compressibility Solver 66
9.1 The S2dTurbSolver Class 66
9.2 The SPC2dSolver Class 68
10 Concluding Remarks 70
References 71

List of Figures

Figure 1: The organization of sub-solvers within the SPC solver. 7

Figure 2: Class hierarchy of SPC Solvers 7

Figure 3: Class hierarchy of Sweeps used by the SPC solvers 8

Figure 4: Class hierarchy for structured blocks 9

Figure 5: Class hierarchy for structured node organizers 10

Figure 6: Restriction of a 3x5 block via the operations restrict_eta(1,4,1) and restrict_eta(0,2,2). 14

Figure 7: A structured two-dimensional block with O topology. 18

Figure 8: A structured two-dimensional block with C topology. 22

Figure 9: A very simple O-block. 25

Figure 10: The matrix corresponding to the O-block of Figure 9 26

Figure 11: The matrix corresponding to the O-block of Figure 9 when an implicit wake cut is used. 27

Figure 12: Class hierarchies of Sweeps and Solvers used by the SPC matrix solver 29

Figure 13: The SPC2dPVSolver part-of hierarchy. 38

Figure 14: Class hierarchy for the turbulence models and sweeps 44

Figure 15: The S2dBLSolver part-of hierarchy 50

Figure 16: The S2dSASolver part-of hierarchy. 57

1 Introduction

At high speeds the noise radiated by a warship is primarily due to cavitation on its propellers. Naval propellers are designed so that cavitation inception is delayed to as high a speed as possible, but the design method requires an accurate prediction of the flow into the propeller plane. Over the past decade computer programs have been developed at DREA to predict the flow into the propeller plane. Known collectively as HLLFLO[1], these programs use a low-order panel method to predict the potential flow, and boundary layer methods to calculate the viscous flow.

The HLLFLO programs have the advantage that they are very fast. They yield fairly good predictions for unappended destroyer hulls[2]; however, there are several areas in which the predictions can be improved.

1. As is typical of boundary layer methods, the wake deficit is over-predicted near the stern. This is especially noticeable at model-scale Reynolds numbers.
2. HLLFLO is incapable of handling the flow past appendages. The wakes shed from the propeller shafts and shaft brackets can cause significant spatially concentrated wake defects which, in turn, affect higher harmonics of the radiated propeller noise and the speed at which cavitation first occurs.
3. HLLFLO does not account for the influence of the boundary layer on the potential flow.

These effects are especially important for smaller vessels with low length to beam ratios. Moreover, HLLFLO is incapable of handling the complex geometries of submarines or advanced marine vehicles such as SWATHs.

It was to address these deficiencies that work on the program TRANSOM was begun. TRANSOM is a Reynolds-averaged Navier-Stokes (RANS) solver: in conjunction with a turbulence model, it solves the Navier-Stokes equations on a grid of points which cover the regions of flow.

When TRANSOM is fully developed it will be applicable to many ship-related flow problems, not just the calculation of propeller inflow: for example, flows around submarines, vortex generation from propellers and control surfaces, bilge vortex generation, and prediction of roll damping coefficients.

The chief practical problem in developing a Navier-Stokes solver for the flow past a ship is that the Reynolds number is so high: for a destroyer moving at 20 knots the Reynolds number is about 10^9. The higher the Reynolds number, the more grid points are necessary to resolve the severe velocity gradients near solid surfaces. For this reason, issues of computer storage are of extreme importance.

Because of the complex geometries of appended surface ships or submarines, the type of grid used in the calculations is also important because it affects both the amount of computer storage required and the level of effort

necessary to generate the grid. Methods which allow the use of unstructured grids (e.g. finite element solvers) typically require far greater computer storage than methods which require structured grids. On the other hand, creating a well-behaved structured grid for a complex geometry can require weeks of effort. Thus, there is a trade-off between the ease generating the grid, and the storage requirements. To address this problem, TRANSOM has been designed as a multi-block multi-method solver. The numerical grid may consist of several blocks, some of which are structured, some unstructured. On each block a solver appropriate to the block structure is used. Since the areas of complex geometry are usually small in comparison with the whole ship, it should be possible to reduce the computer storage requirements significantly.

The current version of TRANSOM includes two different solution methods which can be used on the blocks. The first is based on the finite element program MEF developed at Université Laval[3]; it is suitable for use on unstructured blocks. MEF was modified at DREA to model turbulence using a variant of the k-ε model[4], but the turbulence model has not yet been incorporated into TRANSOM. Details of the implementation of the finite element solver in TRANSOM are given in Reference 5.

The second solution method is based on the program NSI2D developed for DREA under contract to the University of Toronto[6,7,8,9]; it is suitable for use on a single structured block. The algorithm is a variant of the pseudo-compressibility method developed by Rogers and Kwak[10]. NSI2D has recently been extended to three-dimensional flow[11] but the improvements have not yet been incorporated into TRANSOM. Turbulence may be modelled using the Baldwin-Lomax model[12], the Spalart-Allmaras model[13] or the Chen-Patel variant of the k-ε model[14]. The implementation of the pseudo-compressibility solver in TRANSOM is described in detail in this memorandum.

TRANSOM is written in C++ according to the principles of Object Oriented Programming. The main purpose of this memorandum is to describe the C++ classes used to implement the structured pseudo-compressibility (SPC) solver. The numerical algorithms used have been described in detail elsewhere[6–9] and, for the most part, will only be referred to. The overall design of TRANSOM has been described in Reference 15. It is recommended that Reference 15 be read before this memorandum since it describes many of the basic classes used in the SPC solver.

The SPC solver is actually a group of separate solvers each of which can be invoked in isolation or in conjunction with the others. One of these solves for the pressure and velocity given values for the turbulent viscosity. Four others solve for the turbulence viscosity given values for the velocity; they differ in the turbulence model used. Finally, there are two solvers which are used to group the other five together. Four main groups of classes are used to implement these solvers: classes to define the regular structure of structured

blocks, classes to store and solve the linear systems used by the structured solvers, classes to define boundary conditions, and classes used to implement the solvers themselves. After an overview of the pseudo-compressibility method in Section 2, and an overview of the class structure in Section 3, hierarchies for each of these groups of classes are described in Sections 4 to 9. Section 10 contains recommendations for improvements to the program.

TRANSOM input and output files use the OFFSRF format described in Reference 16. Reference 17 describes C++ classes which implement specialized input and output streams for OFFSRF files and a base class which is inherited by all classes which read from or write to OFFSRF files; it should be read prior to this document for a proper understanding of TRANSOM input and output.

2 An Overview of the Pseudo-Compressibility Method

In the pseudo-compressibility method, the equations of fluid motion are written as:

$$\begin{aligned}
&\frac{1}{\beta}\frac{\partial p}{\partial t}+\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0\\
&\frac{\partial u}{\partial t}+\frac{\partial u^2}{\partial x}+\frac{\partial vu}{\partial y}=-\frac{\partial p}{\partial x}+\frac{\partial}{\partial x}\left((\nu+\nu_t)\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left((\nu+\nu_t)\frac{\partial u}{\partial y}\right)\\
&\frac{\partial v}{\partial t}+\frac{\partial uv}{\partial x}+\frac{\partial v^2}{\partial y}=-\frac{\partial p}{\partial y}+\frac{\partial}{\partial x}\left((\nu+\nu_t)\frac{\partial v}{\partial x}\right)+\frac{\partial}{\partial y}\left((\nu+\nu_t)\frac{\partial v}{\partial y}\right)
\end{aligned} \tag{2.1}$$

where u and v are the Cartesian components of the fluid velocity, p is the pressure, ν is the kinematic viscosity, and ν_t is the turbulent viscosity (a Boussinesq approximation of the Reynolds stresses is assumed). An additional term has been added to the equation of continuity in order to convert the equations into a purely hyperbolic system. Steady state solutions to the altered equations are also solutions to the incompressible Navier-Stokes equations since then the added term vanishes. The pseudo-compressibility method integrates the altered equations in time until a steady state is reached.

The turbulent viscosity, ν_t, must be determined using an appropriate turbulence model: see Section 8.1.

The equations are solved on a structured block in which the coordinate directions are denoted ξ and η. On an $m\times n$ block the values of ξ and η range between 0 and $m-1$ and 0 and $n-1$ respectively. The equations of motion are recast in this coordinate system to give:

$$\frac{\partial}{\partial t}D+\frac{\partial}{\partial \xi}(E-E_v)+\frac{\partial}{\partial \eta}(F-F_v)=0 \tag{2.2}$$

where the solution vector, D, the convective fluxes, E and F, and the viscous fluxes, E_v and F_v, are given by:

$$D=J^{-1}\begin{bmatrix}p\\u\\v\end{bmatrix};\quad E=J^{-1}\begin{bmatrix}\beta U\\uU+\xi_x p\\vU+\xi_y p\end{bmatrix};\quad F=J^{-1}\begin{bmatrix}\beta V\\uV+\eta_x p\\vV+\eta_y p\end{bmatrix} \tag{2.3}$$

$$E_v=\nu J^{-1}\begin{bmatrix}0\\g^{11}u_\xi+g^{12}u_\eta-Jv_\eta\\g^{11}v_\xi+g^{12}v_\eta+Ju_\eta\end{bmatrix};\quad F_v=\nu J^{-1}\begin{bmatrix}0\\g^{12}u_\xi+g^{22}u_\eta-Jv_\xi\\g^{12}v_\xi+g^{22}v_\eta+Ju_\xi\end{bmatrix} \tag{2.4}$$

The contravariant velocity components, U and V, are

$$U = \xi_x u + \xi_y v; \quad V = \eta_x u + \eta_y v \tag{2.5}$$

and the Jacobian of the coordinate transformation, J, and the metric coefficients, g^{ij}, are given by:

$$J^{-1} = \frac{\partial x}{\partial \xi}\frac{\partial y}{\partial \eta} - \frac{\partial x}{\partial \eta}\frac{\partial y}{\partial \xi} \tag{2.6}$$

$$\xi_x = J\frac{\partial y}{\partial \eta}; \quad \xi_y = -J\frac{\partial x}{\partial \eta}; \quad \eta_x = -J\frac{\partial y}{\partial \xi}; \quad \eta_y = J\frac{\partial x}{\partial \xi} \tag{2.7}$$

$$g^{11} = \xi_x^2 + \xi_y^2; \quad g^{12} = g^{21} = \xi_x\eta_x + \xi_y\eta_y; \quad g^{22} = \eta_x^2 + \eta_y^2 \tag{2.8}$$

Equation 2.2 may be written

$$\frac{\partial}{\partial t}D = -R \tag{2.9}$$

with

$$R = \frac{\partial}{\partial \xi}(E - E_v) + \frac{\partial}{\partial \eta}(F - F_v) \tag{2.10}$$

Equation 2.9 is solved using an implicit first order discretization scheme in time. Letting a subscript n denote a value at time step n, we have:

$$\frac{\Delta D}{\Delta t} = -R(D_{n+1}) \tag{2.11}$$

with $\Delta D = D_{n+1} - D_n$. The value of R at time step $n+1$ is obtained from R_n using a Taylor expansion:

$$R_{n+1} \approx R_n + \Delta D\frac{\partial R_n}{\partial D} \tag{2.12}$$

The linear system to be solved at each time step is then:

$$\left(\frac{1}{\Delta t} + \frac{\partial R_n}{\partial D}\right)\Delta D = -R_n \tag{2.13}$$

Details of the spatial discretization scheme are given in Reference 6.

3 An Overview of the Class Structure Used for SPC Solvers

In this section we present a brief overview of the hierarchy of classes which represent the solvers used in the pseudo-compressibility method. The TRANSOM SPC solver contains five principal sub-solvers.

1. A solver which, given values for the turbulent viscosity, ν_t, solves Equation (2.1) for the pressure and velocity. This solver is represented by the class SPC2dPVSolver described in Section 7.2.
2. A solver which updates the turbulent viscosity for laminar flow. This solver is trivial, simply setting the turbulent viscosity to zero. It is defined so that it may be included as a sub-solver in a sequential solver: see Reference 15, Section 4.4.2. This solver is represented by the class LaminarSolver.
3. A solver which updates the turbulent viscosity using to the Baldwin-Lomax turbulence model[12]. This solver is represented by the class S2dBLSolver described in Section 8.2.3.
4. A solver which updates the turbulent viscosity using to the Spalart-Allmaras turbulence model[13]. This solver is represented by the class S2dSASolver described in Section 8.3.4.
5. A solver which updates the turbulent viscosity using to the Chen-Patel variant of the k-ε turbulence model[14]. This solver is represented by the class S2dCPSolver described in Section 8.4.2.

Each of these solvers is derived from the base class Solver described in Reference 15, Section 4.3. The classes SPC2dPVSolver, S2dCPSolver, and S2dSASolver are derived from the class ResidualSolver, a specialization of Solver which represents solvers which operate by minimizing a residual such as the right hand side of Equation (2.9). ResidualSolver is described in Reference 15, Section 4.6.3.

The class S2dTurbSolver is a sequential solver (represented by the class SeqSolver: see Reference 15, Section 4.4.2) which allows the four turbulence solvers to be executed in sequence, the laminar solver initializing the Baldwin-Lomax solver which initializes the Spalart-Allmaras solver which initializes the Chen-Patel solver. Any subset of the four solvers may be used.

The class SPC2dSolver runs an SPC2dPVSolver and an S2dTurbSolver concurrently (see Reference 15, Section 4.4.1 for a description of concurrent solvers and the ConcSolver class). Thus it solves for the pressure, velocity, and turbulent viscosity concurrently.

The arrangement of the sub-solvers within an SPC2dSolver is shown in Figure 1. In this figure the horizontal arrows denote solvers executed concurrently while the vertical arrows denote solvers executed sequentially. The full inheritance hierarchy of SPC solver classes is shown in Figure 2.

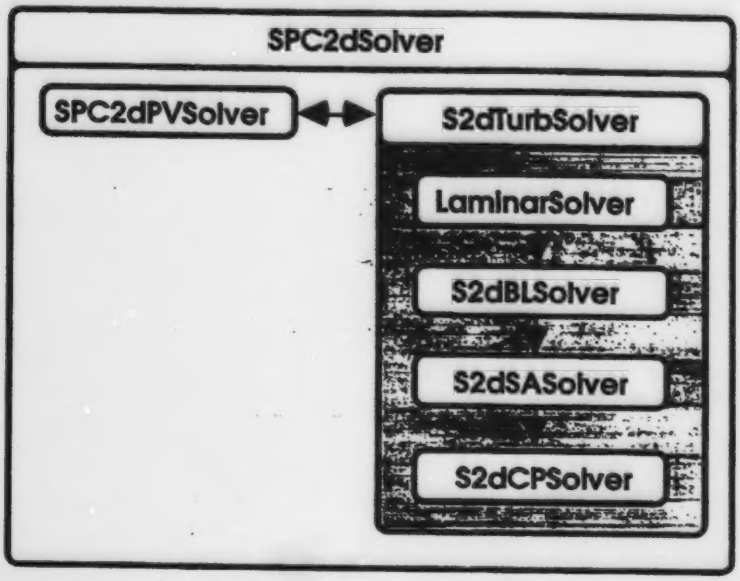


Figure 1: *The organization of sub-solvers within the SPC solver.*

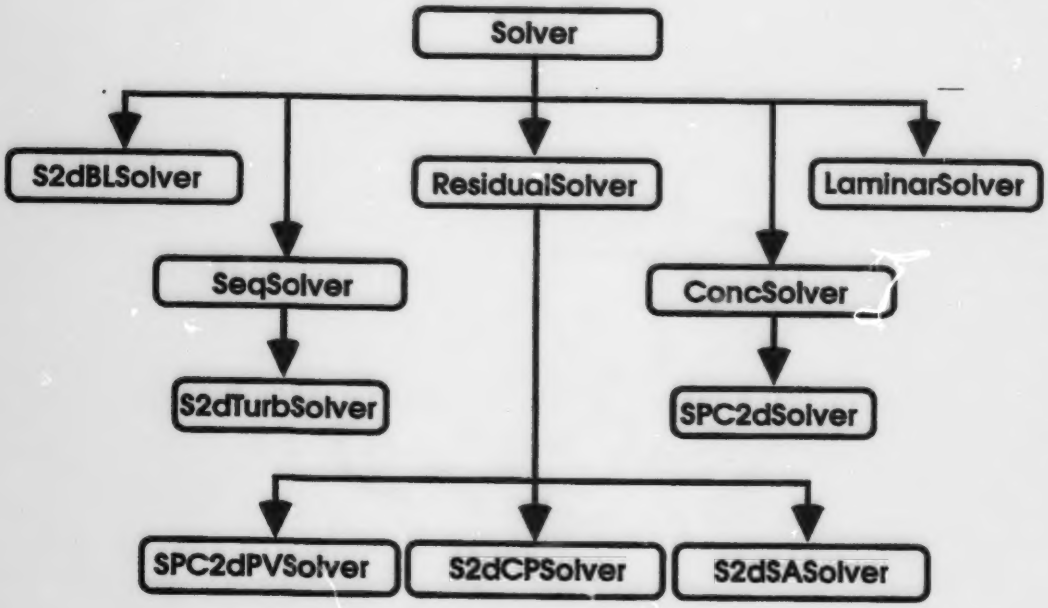


Figure 2: *Class hierarchy of SPC Solvers*

A Solver represents the iterative solution of some equation or set of equations. Every Solver consists of two parts: a sweep, which updates the independent variables at each iteration; and a stop test, which determines when the iteration should stop. These are represented by the classes Sweep and StopTest described in Reference 15, Sections 4.1 and 4.2. The hierarchy of sweeps used by the SPC solvers is shown in Figure 3. This hierarchy closely mirrors that for the solvers, except that there are no sweeps corresponding to LaminarSolver, S2dTurbSolver and SPC2dSolver. LaminarSolver does not require a sweep having set the turbulent viscosity to zero during initialization. S2dTurbSolver and SPC2dSolver use the SeqSweep and ConcSweep respectively for their sweeps.

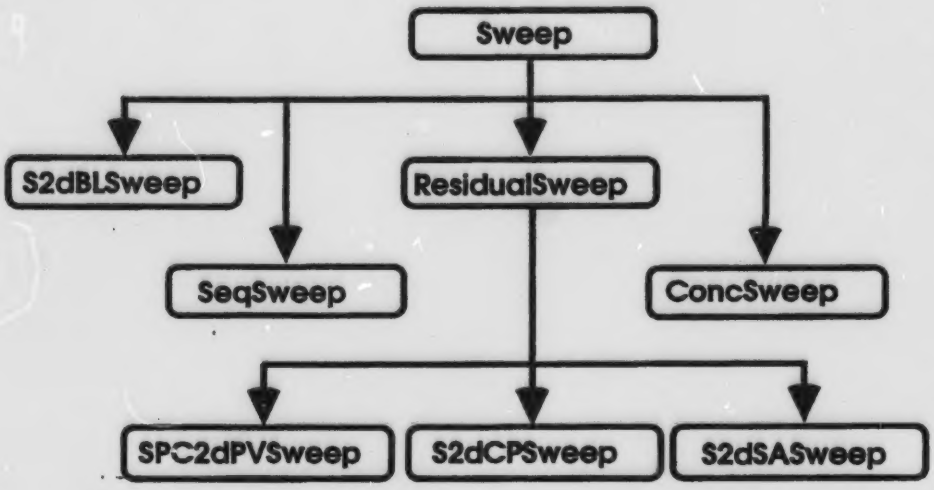


Figure 3: *Class hierarchy of Sweeps used by the SPC solvers*

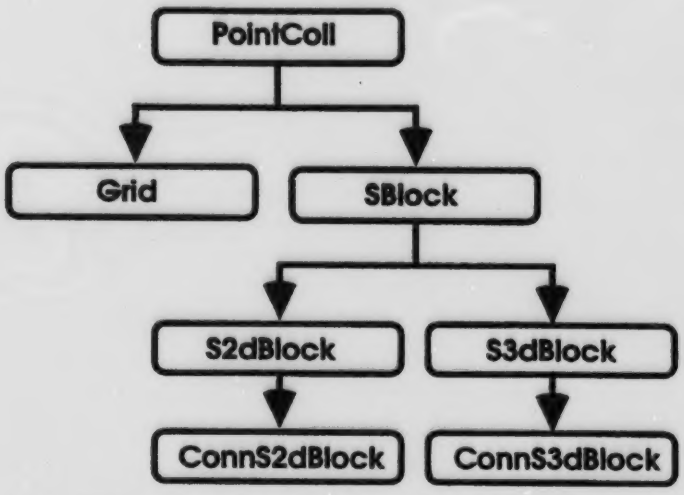


Figure 4: *Class hierarchy for structured blocks*

4 Structured Blocks

A structured block is one whose nodes can be organized into a rectangular array: $m_1 \times m_2$ in two dimensions, $m_1 \times m_2 \times m_3$ in three dimensions. Figure 4 shows the class hierarchy for the classes S2dBlock and S3dBlock which represent simple structured blocks of two and three dimensions respectively. Only the S2dBlock class will be described here; the S3dBlock class is similar but has not yet been fully implemented. The class ConnS2dBlock is a specialization of S2dBlock which allows re-entrant boundaries. It is described in Section 4.3.

The nodes of an $m_1 \times m_2$ structured block may be indexed using the pair (i, j), where i is in the range 0 to $m_1 - 1$ and j is in the range 0 to $m_2 - 1$. To help in distinguishing between node numbers and indices, TRANSOM uses the type Node for nodes, and XiIndex and EtaIndex for the indices i and j, respectively. In the current implementation Node is equivalent to the type int while XiIndex and EtaIndex are equivalent to the type unsigned int. Because some data structures cannot distinguish between the ξ and η directions (e.g. the class S1dNodeOrg: see below) the generic boundary index Index is also defined. It is equivalent to either an XiIndex or a EtaIndex.

4.1 Structured Node Organizers

As described in Reference 15, TRANSOM recognizes the organization of nodes in a block as an important abstraction independent of the nodes themselves. Node organizers are specializations of the base class NodeOrg. The class hierarchy for the structured node organizers is shown in Figure 5.

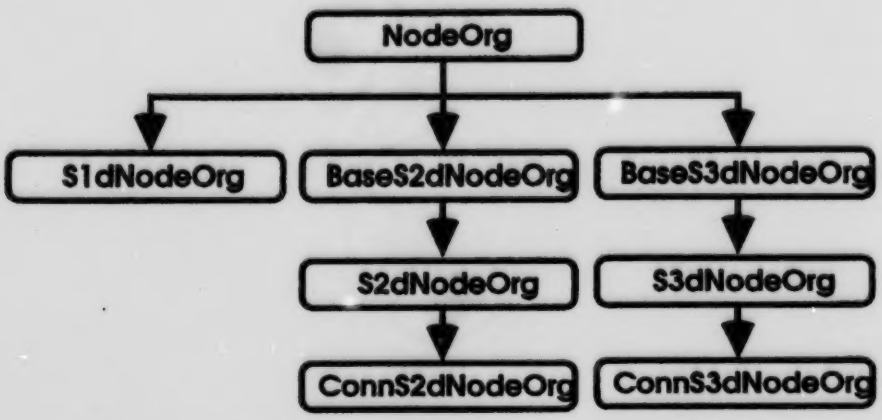


Figure 5: *Class hierarchy for structured node organizers*

4.1.1 The BaseS2dNodeOrg Class

The class BaseS2dNodeOrg organizes the nodes of a structured two-dimensional block. For convenience we label the coordinate directions ξ and η. Thus, an $m_1 \times m_2$ block has m_1 nodes in the ξ direction and m_2 nodes in the η direction.

The BaseS2dNodeOrg class has the following constructors and member functions.

BaseS2dNodeOrg(unsigned m1, unsigned m2);
Creates a node organizer for an m1 x m2 structured block.

BaseS2dNodeOrg(unsigned m1, unsigned m2, const Node &first);
Creates a node organizer for an m1 x m2 structured block. The first node in the block is first. This constructor is used for creating a structured block which shares the node numbering of the grid of which it is a part. For example, if we wish to organize nodes 100 to 199 of a grid into a 10×10 block while maintaining the numbering 100 to 199, we would use the following code:

```
BaseS2dNodeOrg norg(10,10,100);
```

The alternative,

```
BaseS2dNodeOrg norg(10,10);
```

would create a block with nodes numbered from 0 to 99.

unsigned nxi() const;
unsigned neta() const;
Return the number of nodes in the ξ and η directions, respectively.

virtual int contains(Node n) const;
Returns true if the block contains node n.

void transpose();

Transposes the nodes so that the ξ direction becomes the η direction: node (i,j) becomes node (j,i).

Node operator()(XiIndex i, EtaIndex j) const;

Returns the node whose indices are (i,j). Thus, if norg is a BaseS2dNodeOrg, then norg(i,j) returns the node whose indices are (i,j).

virtual AllNodeIter *get_all_node_iter();

Returns an iterator over all the nodes in the block. See Reference 15, Section 2.2 for a description of the class AllNodeIter.

SNodeIter xi_node_iter(EtaIndex j = 0) const;

Returns an iterator over the nodes in row j.

SNodeIter eta_node_iter(XiIndex i = 0) const;

Returns an iterator over the nodes in column i.

The last two member functions return iterators of type SNodeIter. This is a class which iterates over a row or column of a structured block. The constructor for this class is private; it can only be constructed using the BaseS2dNodeOrg member functions xi_node_iter() and eta_node_iter(). As well as the standard iterator member functions, an SNodeIter has the following member functions for changing the row over which the iteration occurs.

Node next_row();

Resets the iterator to the beginning of the next row or column. Returns that node. If the row or column is out of range, then a call to in_range() will return false.

Node prev_row();

Similar to next_row() but resets the iterator to the beginning of the previous row or column.

void set_to_node(const Node &n);

Sets the current node to n. Adjusts the range of the iterator to be the row or column containing n.

The iterator returned by the member function get_all_node_iter() is of type S2dAllNodeIter, a specialization of AllNodeIter for iterating over a structured block. Like the SNodeIter, its constructor is private so that it may only be obtained via BaseS2dNodeOrg::get_all_node_iter(). It uses the standard TRANSOM iterator syntax and includes a set_to_node() member function similar to that of SNodeIter.

The following code uses three different iteration methods to set the value of the pressure to 1.0 at every node of a structured block.

```
BaseS2dNodeOrg norg(50,60);
PVIOVar pv(norg);

// Method 1: Use AllNodeIter
AllNodeIter *iter1 = norg.get_all_node_iter();
while(iter1->in_range()) {
  pv.p(iter1) = 1.0;
  iter1->next();
}
delete iter1;

// Method 2: Use row iterator
SNodeIter *iter2 = norg.xi_node_iter();
while(iter2->in_range()) {
  while(iter2->in_range()) {
    pv.p(iter1) = 1.0;
    iter1->next();
  }
  iter2->next_row();
}
delete iter2;

// Method 3: Use indices
for (EtaIndex j = 0; j < norg.neta(); j++)
  for (XiIndex i = 0; i < norg.nxi(); i++)
    pv.p(norg(i,j)) = 1.0;
```

In this case Method 1 is the most efficient; however, the AllNodeIter is not so useful when only a subset of the nodes is to be used or if special cases must be handled at boundary nodes. Method 3 is convenient to code but is relatively costly since two multiplications and two additions are required to compute the function norg(i,j).

4.1.2 The S2dNodeOrg Class

An S2dNodeOrg is a BaseS2dNodeOrg which also allows restrictions to sub-blocks. The restrictions are stored on a stack so that the node organizer can later be "unrestricted".

The S2dNodeOrg constructors are the same as those of the BaseS2dNodeOrg and it inherits all the BaseS2dNodeOrg member functions. In addition it has the following member functions for restricting the block.

```
void restrict_xi(XiIndex mn, XiIndex mx, unsigned s);
void restrict_eta(EtaIndex mn, EtaIndex mx, unsigned s);
```

Restrict the nodes in the ξ or η directions to be the nodes between mn and mx with a stride of s.

```
void unrestrict();
```

Resets to the previous restriction.

```
void unrestrict_all();
```

Resets the node organizer to the unrestricted nodes.

As an example of block restriction, consider a 3×5 block with nodes numbering 0 through 14: see Figure 6. The call restrict_eta(1,3,1) causes the block to be restricted to the 3×3 sub-block with j values 1, 2, and 3: i.e. to nodes 3 to 11. On the new sub-block, the numbering of the nodes remains the same but the j indexing is changed to range from 0 to 2. Further restrictions must be made using the new (i,j) indexing. Thus, a second restriction using the call restrict_eta(0,2,2) yields a 3×2 sub-block with nodes 3, 4, 5, 9, 10, and 11.

This example is reflected in the following code in which the node organizer is created, then restricted twice, then unrestricted. The node (0,1) is printed after each restriction.

```
S2dNodeOrg norg(3,5);
cout << norg(0,1);
norg.restrict_eta(1,3,1);
cout << ' ' << norg(0,1);
norg.restrict_eta(0,2,2);
cout << ' ' << norg(0,1);
norg.unrestrict();
cout << ' ' << norg(0,1);
norg.unrestrict();
cout << ' ' << norg(0,1) << endl;
```

The output is:

```
3 6 9 6 3
```

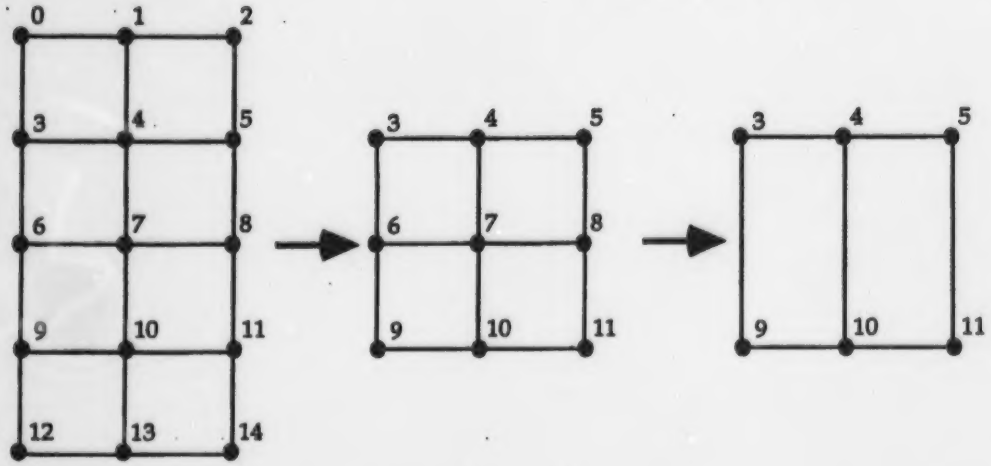
Notice how the node referred to as (0,1) changes as the block is restricted and unrestricted.

Although block restrictions have not yet been applied much in TRANSOM, they have several important uses. First, they are a convenient means for performing grid refinement studies, an important procedure in all flow simulations. Second, they provide a means for implementing convergence acceleration through successive grid refinement and through multigrid methods. Although multigrid methods have not yet been implemented in TRANSOM, preliminary studies have been performed by Royal[18]. Third, they provide a convenient means for partitioning a structured block into sub-blocks to which different solvers can be applied.

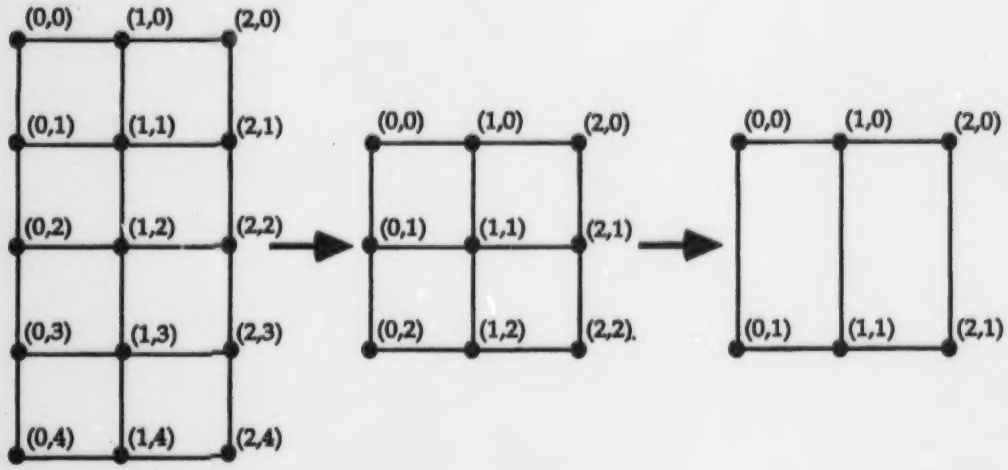
The S2dNodeOrg class also provides the following member functions to support the creation of data structures which store values for the nodes on a single row of a block.

```
const S1dNodeOrg& xi_array_row_node_org() const;
const S1dNodeOrg& eta_array_row_node_org() const;
```

The class S1dNodeOrg reprsents a one-dimensional structured node organizer. It is discussed in the next section. For example, suppose we wished to store the node numbers along the row $j=5$ of a 15×20 block. We could do so as follows:



a) Node numbering



b)(i,j) indexing

Figure 6: *Restriction of a 3x5 block via the operations restrict_eta(1,4,1) and restrict_eta(0,2,2). Figure a) shows that the node numbers remain invariant while Figure b) shows that the (i,j) indexing does not.*

```
S2dNodeOrg norg(15,20);
S1dNodeOrg &rnorg = norg.eta_array_row_node_org();
NodeVar<Node> nodes_j5(rnorg);
for (XiIndex i = 0; i < rnorg.nxi(); i++)
  nodes_j5[i] = norg(i,5);
```

The array nodes_j5 now stores the nodes 65 through 79. The NodeVar created in this fashion will be restricted when the original block is restricted. That is, if we followed the preceding code with:

```
norg.restrict_eta(0,8,2);
AllNodeIter *n = nodes_j5.get_node_org()->get_all_node_iter();
while (n->in_range()) {
  cout << nodes_j5[*n] << ' ';
cout << endl;
delete n;
```

The output would be:

```
65 67 69 71 73
```

and not

```
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
```

S2dNodeOrg also provides a means for referring to each of the four boundaries of the block.

```
enum BndType { xilo, xihi, etalo, etahi };
```

These refer to the boundaries with indices i = 0, i = nxi()-1, j = 0, and j = neta()-1, respectively.

4.1.3 The S1dNodeOrg Class

An S1dNodeOrg is a one-dimensional structured node organizer. The nodes in a two-dimensional block can be referred to either by node number or by the indices (i, j). Similarly, the nodes governed by an S1dNodeOrg can be referred to by node number or by an index i. However, with an S1dNodeOrg there is more cause for confusion, since both the index and the node number can both be represented as integers. The index, i, always ranges between 0 and $m-1$, where m is the number of nodes governed by the node organizer. The index of a node changes when the node organizer is restricted. On the other hand, the node numbers range between zero and the maximal number of nodes governed by the node organizer; not all the numbers in this range need be used. Moreover, the node numbers do not change when the node organizer is restricted. When using an S1dNodeOrg it is very important to maintain a clear distinction between arguments of type Node and arguments of type Index.

The S1dNodeOrg class has the following constructors and member functions:

S1dNodeOrg(unsigned m);
Makes an S1dNodeOrg for m nodes. The node numbers are in the range 0 to m .

S1dNodeOrg(unsigned m, const Node &f);
Makes an S1dNodeOrg for m nodes. The number of the first node is f, so that the node numbers are in the range f to f+m-1.

virtual int contains(const Node &n) const;
True if n is in the collection of nodes.

void restrict(Index mn, Index mx, unsigned s);
Restricts the nodes.

void unrestrict();
Resets to the previous restriction.

void unrestrict_all();
Resets to the unrestricted nodes.

virtual AllNodeIter* get_all_node_iter() const;
Returns an iterator over all the nodes of the current restriction.

Node operator()(Index i) const;
Return i^{th} node in the restricted numbering.

The class RNodeIter may be used to iterate over the nodes governed by an S1dNodeOrg. It is a non-virtual (and hence somewhat faster) alternative to the AllNodeIter returned by RNodeIter::get_all_node_iter(). The prototype for its constructors are:

RNodeIter(const S1dNodeOrg &r);
Makes an RNodeIter for the nodes in r.

RNodeIter(const S1dNodeOrg &r, Node c);
Makes an RNodeIter for the nodes in r but starting at the node c.

The standard syntax for TRANSOM iterators is used (see Reference 15, Section 2.2).

4.1.4 The ConnS2dNodeOrg Class

A ConnS2dNodeOrg is an S2dNodeOrg with additional data structures to represent re-entrant boundaries such as those encountered in O and C topology grids. These data structures have the same form, in C syntax, as the Fortran arrays bndmap, bndinv, and bndfluxmap used in NSI2D and described in Reference 7. This allows these arrays to be used both in C and Fortran routines so that many of the NSI2D subroutines can be re-used within TRANSOM.

As well as the member functions inherited from the S2dNodeOrg class, ConnS2dNodeOrg has the following constructors and member functions:

ConnS2dNodeOrg(unsigned m1,unsigned m2);
Makes a node organizer for an m1 x m2 block. By default there will be no connected boundaries. The maximal number of nodes will be

m1*m2.

ConnS2dNodeOrg(unsigned m1,unsigned m2,const Node &f);
Makes a node organizer for an m1 x m2 block whose first node is f. The maximal number of nodes will be f + m1*m2.

virtual Str topology() const;
Returns the string "S" to indicate neither "O" nor "C" topology.

void transpose();
Transposes the block so that index (i,j) becomes (j,i)

Node& get_bndmap(Index n, unsigned j, BndType bnd);
Node get_bndmap(Index n, unsigned j, BndType bnd) const;
Returns the j^{th} node across the boundary from the node with index n on boundary bnd. For example, for the block shown in Figure 7, node 9 has indices $(i,j) = (0,1)$. The function get_bndmap(1,1,xilo) would return node 16.

int& get_bndinv(Index n, BndType bnd);
int get_bndinv(Index n, BndType bnd) const;
Returns the inversion flag for the n^{th} node on boundary bnd: see Reference 7 for a description of the inversion flags.

4.2 The S2dBlock Class

The S2dBlock class represents a structured two-dimensional block. It is implemented as a PointColl (see Reference 15, Section 2.4) whose nodes are organized by an S2dNodeOrg. There are four S2dBlock constructors.

S2dBlock(unsigned m1, unsigned m2);
Creates a structured m1 x m2 block.

S2dBlock(unsigned m1, unsigned m2, const Node &f, double *d);
Creates a structured m1 x m2 block whose first node is node f and whose coordinates are stored in memory beginning at d. This constructor is used for creating a structured block which shares both the coordinates and the node numbering of the grid of which it is a part. For example, if we wish the nodes 100 to 199 of a 300 node grid to be a 10×10 block which maintains the node numbering 100 to 199, we would use the following code:

```
Grid grid(2,300);
S2dBlock sblock(10,10,100,&grid[0]);
```

S2dBlock(const S2dBlock &s);
Creates a structured block by copying s.

S2dBlock(const NodeOrg &norg, const S2dBlock &s, const Node &n);
Creates an S2dBlock governed by norg which shares the coordinates of s starting at node n. This constructor is used primarily by specializations of S2dBlock to construct blocks with specialized node organizers. It is protected.

The S2dBlock class defines the following member functions.

```
unsigned nxi() const;
unsigned neta() const;
```
Return the ξ and η dimensions of the block, respectively.

```
S2dNodeOrg* get_node_org() const;
```
Returns a pointer to the node organizer governing the block;

In addition it inherits the following member functions from its base class PointColl:

```
unsigned num_nodes() const;
```
Returns the number of nodes in the block.

```
unsigned dim() const;
```
Returns the dimension (one, two, or three) of the points.

```
const Str& get_name() const;
```
Returns the name of the block (Str is a class representing character strings).

```
double* operator[](Node);
```
Returns a pointer to the values of the coordinates. Thus, if p is a S2dBlock, then p[n][0] is the x-value of the coordinates at node n, p[n][1] is the y-value, and p[n][2] is the z-value.

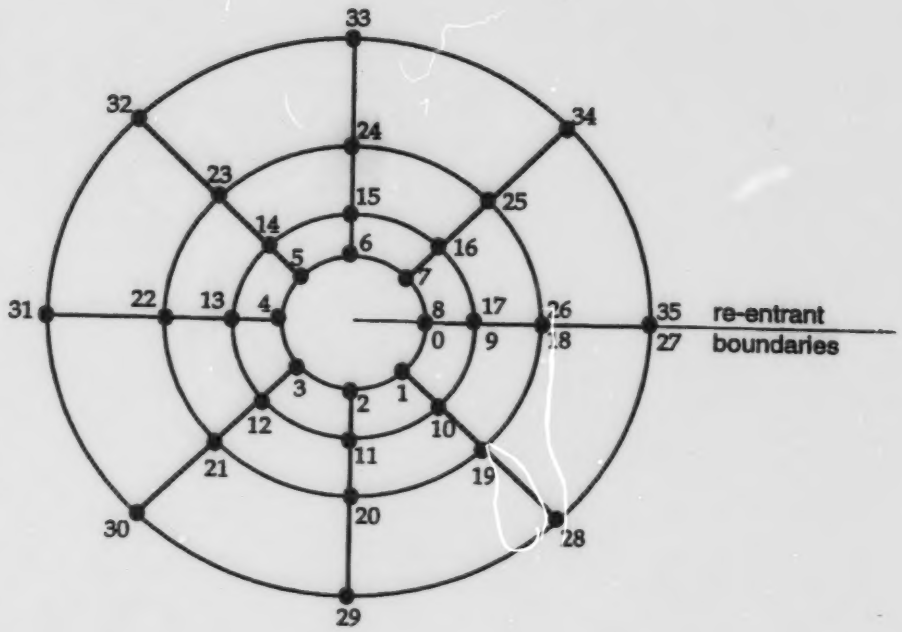


Figure 7: *A structured two-dimensional block with O topology.*

4.3 The ConnS2dBlock Class

An ConnS2dBlock is an S2dBlock whose nodes are organized using an ConnS2dNodeOrg. Thus, it contains the coordinates of the block as well as information on the connectivity of its boundaries.

The class ConnS2dBlock also stores the gradients of the coordinates, $\nabla\xi$ and $\nabla\eta$; the Jacobian, J; and the metric functions g^{11}, g^{12}, and g^{22}. They are represented by the struct Metric2d which has the following data members (the classes NumNodeVar and MultiNumVar are described in Reference 15, Sections 3.2 and 3.3 respectively).

MultiNumVar *gradxi, *gradeta;
: Storage for the gradients of ξ and η scaled by the Jacobian.

NumNodeVar *jacc;
: The Jacobian, J, at each node.

NumNodeVar *g11, *g12, *g22;
: The metrics g^{11}, g^{12}, and g^{22} at each node.

The layout in memory for all of these variables is equivalent to that used by NSI2D so that they can be passed directly to Fortran subroutines imported by

NSI2D. The metric variables are stored as pointers so that memory need be allocated for them only when necessary.

The metrics in a ConnS2dBlock are stored as a public data member, metrics, of type Metric2d. This lack of encapsulation of the metrics is a design flaw which will be remedied is future versions of TRANSOM by providing member functions for access to the metric values. In the current version of TRANSOM, the metrics are used only within Fortran subroutines imported from NSI2D so that the lack of encapsulation is not serious.

As well as the member functions inherited from S2dBlock, ConnS2dBlock has the following constructors and member functions:

ConnS2dBlock(unsigned m1, unsigned m2, Node f = 0, double *d = 0);
Constructs a structured block with dimensions m1 x m2 and whose first node is f. If d is not null, storage for the coordinates will start at d; otherwise new memory will be allocated.

ConnS2dBlock(ConnS2dNodeOrg *norg, double *d = 0);
Constructs a structured block using node organizer norg. If d is not null, storage for the coordinates will start at d; otherwise new memory will be allocated. This constructor is used primarily by specializations of ConnS2dBlock whose node organizers are specializations of ConnS2dNodeOrg.

ConnS2dBlock(const S2dBlock& b, ConnS2dNodeOrg *norg = 0);
Constructs a block by copying b and sharing its coordinates, but giving it the node organizer norg. If norg is null, the node organizer from b is copied. The primary use of this constructor is to promote an S2dBlock to a ConnS2dBlock.

ConnS2dNodeOrg* get_node_org() const;
Returns a pointer to the node organizer for the block.

void calc_metrics(int flag);
Calculates $\nabla\xi$ and $\nabla\eta$, J, g^{11}, g^{12}, and g^{22}. If flag is false do not recalculate if the metrics have already been calculated; if true, always calculate them. If memory has not yet been allocated for the metrics, it will be allocated now.

void transpose();
Transposes the block by transposing the node organizer and swapping $\nabla\xi$ and $\nabla\eta$, and g^{11} and g^{22}.

Str topology();
Returns the string "S", "O", or "C" to describe the topology.

In the TRANSOM input file an ConnS2dBlock can be defined within a GRID record using a STRUCTURED BLOCK record as follows:

{GRID: *ndim npts*
{STRUCTURED BLOCK: *nxi neta first-node*
{NAME: *name* }
{COORDINATES
x_1 y_1 z_1
x_2 y_2 z_2
⋮ ⋮ ⋮
x_n y_n z_n
}COORDINATES
}STRUCTURED BLOCK
More records to specify blocks
}GRID

The dimensions of the grid and its number of nodes are specified by *ndim* and *npts*, respectively (see Reference 15, Section 2.5). The dimensions of the block are given by *nxi* and *neta* and the number of its first node by *first-node*. If *first-node* is absent, the first node of the block will be the first node of the grid which has not yet been assigned to a block. As explained in Reference 15, Section 2.5, the classes Grid and S2dBlock are both able to read coordinate values from a COORDINATES record. This means that one may choose where to define the coordinate values in the input file. They can all be specified in the GRID record or the points for each block may be specified within the STRUCTURED BLOCK records defining the blocks.

4.4 Blocks with O Topology

Figure 7 shows a simple example of a structured block with O topology. The boundaries xilo and xihi are joined to form a single re-entrant boundary. Structured blocks with O topology may be represented in TRANSOM using the classes OTopS2dNodeOrg and OTopS2dBlock. It should be noted that these classes provide no extra functionality that is not already present in the more general classes ConnS2dNodeOrg and ConnS2dBlock. They are merely a convenience.

The class OTopS2dNodeOrg is simply an ConnS2dNodeOrg with O topology. The connecting boundaries are xilo and xihi. The constructors are similar to those of the ConnS2dNodeOrg.

OTopS2dNodeOrg(unsigned m1,unsigned m2);
Makes a node organizer for an m1 x m2 O-block.

OTopS2dNodeOrg(unsigned m1,unsigned m2,const Node &f);
Makes a node organizer for an m1 x m2 O-block whose first node is f.

virtual Str topology() const;
Returns the string "O".

All other member functions are inherited from ConnS2dNodeOrg.

The class OTopS2dBlock is simply an ConnS2dBlock whose node organizer is a OTopS2dNodeOrg. Its constructors are similar to those of the ConnS2dBlock.

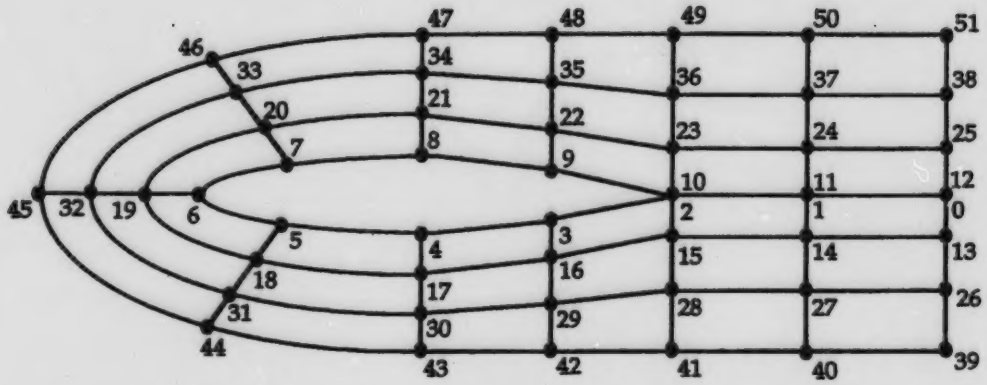


Figure 8: *A structured two-dimensional block with C topology.*

OTopS2dBlock(unsigned m1, unsigned m2, double *d = 0);

Constructs an O-block with dimensions m1 x m2. If d is not null, storage for the coordinates will start at d; otherwise new memory will be allocated.

OTopS2dBlock(unsigned m1, unsigned m2, const Node &f, double *d = 0);

Constructs an O-block with dimensions m1 x m2 and whose first node is f. If d is not null, storage for the coordinates will start at d; otherwise new memory will be allocated.

OTopS2dBlock(S2dBlock& b);

Construct the block by copying b and sharing its coordinates, but giving it an OTopS2dNodeOrg.

In the TRANSOM input file an OTopS2dBlock can be defined within a GRID record using an O BLOCK record which is essentially identical to a STRUCTURED BLOCK record:

{GRID: *ndim npts*
 {O BLOCK: *nxi neta first-node*
  {NAME: *name* }
  {COORDINATES
   x_1 y_1 z_1
   x_2 y_2 z_2
   ⋮ ⋮ ⋮
   x_n y_n z_n
  }COORDINATES
 }O BLOCK
 More records to specify blocks
}GRID

4.5 Blocks with C Topology

Figure 8 shows a simple example of a structured block with C topology. The start and end of the boundary etalo are joined to form a re-entrant portion. Structured blocks with C topology may be represented in TRANSOM using the classes CTopS2dNodeOrg and CTopS2dBlock. As with the O topology, these classes are merely a convenience; they provide no functionality not already present in the classes ConnS2dNodeOrg and ConnS2dBlock.

The class CTopS2dNodeOrg is simply an ConnS2dNodeOrg with C topology. The self-connecting boundary is etalo. The constructors are similar to those of the ConnS2dNodeOrg but with extra arguments to define the region of the boundary connection.

```
CTopS2dNodeOrg(unsigned m1, unsigned m2, XiIndex i1, XiIndex i2);
```
Makes a node organizer for an m1 x m2 C-block. The non-connecting part of the boundary is between the nodes with indices i1 and i2, inclusive.

```
CTopS2dNodeOrg(unsigned m1, unsigned m2, const Node &f,
                XiIndex i1,XiIndex i2);
```
Makes a node organizer for an m1 x m2 C-block whose first node is f. The non-connecting part of the boundary is between the nodes with indices i1 and i2, inclusive.

```
virtual Str topology() const;
```
Returns the string "C".

All other member functions are inherited from ConnS2dNodeOrg.

The class CTopS2dBlock is simply an ConnS2dBlock whose node organizer is a CTopS2dNodeOrg. It has the following constructors.

```
CtopS2dBlock(unsigned m1, unsigned m2, XiIndex i1, XiIndex i2, double *d = 0);
```
Constructs a C-block with dimensions m1 x m2, non-connecting region i1 to i2, and memory at d.

```
CTopS2dBlock(unsigned m1, unsigned m2, const Node &f, XiIndex i1, XiIndex i2,
             double *d = 0);
```
Constructs a C-block with dimensions m1 x m2, first node f, non-connecting region i1 to i2, and memory at d.

```
CTopS2dBlock(const S2dBlock& b, XiIndex i1, XiIndex i2);
```
Constructs a C-block by copying b and sharing its coordinates, but giving it a CTopS2dNodeOrg. The non-connecting part of the boundary is between the nodes with indices i1 and i2, inclusive.

All other member functions are inherited from ConnS2dBlock.

In the TRANSOM input file a CTopS2dBlock can be defined within a GRID record using an C BLOCK record which is similar to a STRUCTURED BLOCK or O BLOCK record:

{GRID: *ndim npts*
{C BLOCK: *nxi neta itail1 itail2 first-node*
{NAME: *name* }
{COORDINATES

x_1	y_1	z_1
x_2	y_2	z_2
⋮	⋮	⋮
x_n	y_n	z_n

}COORDINATES
}C BLOCK
More records to specify blocks
}GRID

The parameters *itail1* and *itail2* specify the location of the non-connecting part of the boundary etalo.

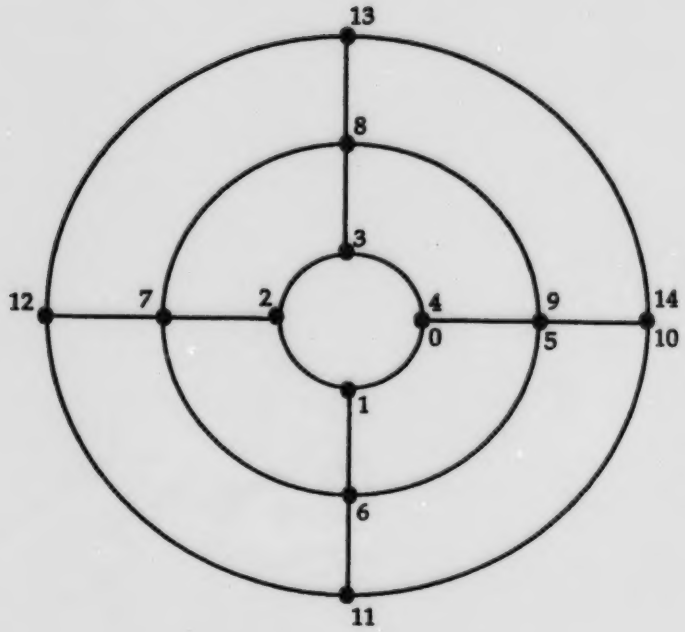


Figure 9: *A very simple O-block.*

5 Classes to Represent and Solve the Linear System

At each time step the linear system of Equation (2.13) must be solved. At each node, the derivative of the residuals, $\partial R / \partial D$, is approximated using the values of D at the node and its four neighbours. This leads to a matrix which has five principal bands, denoted A, B, C, D, and E. These bands represent the dependence of a node (i,j) on the neighbouring nodes $(i,j-1)$, $(i-1,j)$, (i,j), $(i+1,j)$, and $(i,j+1)$ respectively. Each element of the bands is a 3×3 matrix. Each band is stored as an array of 3×3 matrices with length N, where N is the number of nodes in the block.

The turbulence solvers require matrices of similar structure but they have only one or two independent variables per node. Hence, they require the elements of the matrices to be either a single floating point value or a 2×2 matrix. The TRANSOM matrices have been defined so that the matrix elements can be 1×1, 2×2, or 3×3 matrices. For the remainder of this discussion we will assume that they are 3×3 matrices.

When a node lies on one of the boundaries of the block, one of its neighbours does not exist; the element of the band corresponding to this node is disregarded. Alternatively, on a re-entrant boundary of an O-block or a C-block, the neighbouring node exists but in a different part of the block. For example, on the 5×3 O-block of Figure 9 the nodes 0 and 3 are neighbours, as

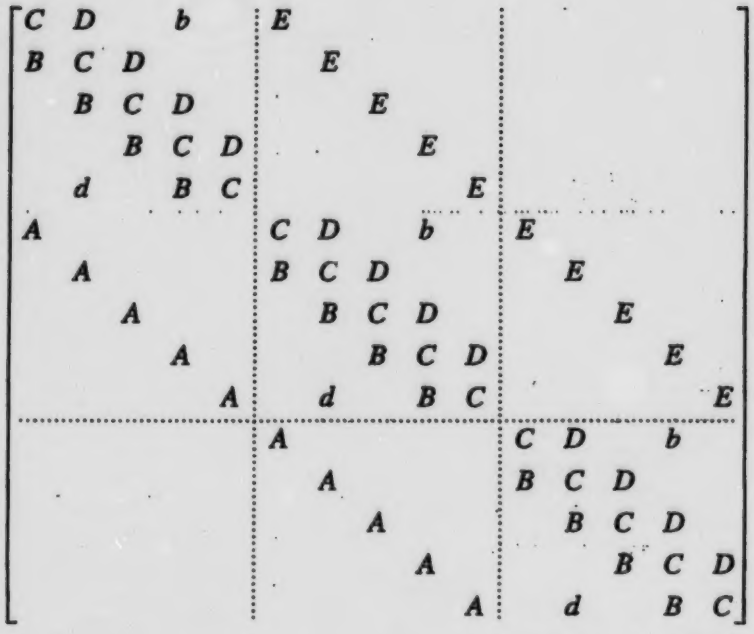


Figure 10: *The matrix corresponding to the O-block of Figure 9*

are 5 and 8, and 10 and 13. Similarly, the nodes 1, 6, and 11 are neighbours of 4, 9, and 14 respectively. The structure of the matrix corresponding to this block is shown in Figure 10. The lower case letters denote the elements corresponding to neighbours of nodes on re-entrant boundaries.

The TRANSOM SPC solvers also allow the use of an "implicit wake cut" to ensure that the values of the independent variables at duplicated nodes of a re-entrant boundary have the same values. The technique used is detailed in Reference 11. For the case of the simple 5×3 O-block of Figure 9, the nodes 3, 8, and 13 remain as neighbours to 0, 5, and 10, respectively. However, the value of the duplicated node 4 is tied to the value of node 0; hence, the fourth row of the matrix has only two elements: (4,0) and (4,4). The nodes 9 and 14 are treated similarly. The resulting matrix structure is shown in Figure 11. The off-diagonal elements in rows 4, 9, and 14 are stored in the array containing diagonal D and so they are shown with a lower case d.

5.1 The SPCMtx Class

The matrix representation used by TRANSOM is essentially the same as that used by NSI2D. In the C++ code the representation is encapsulated in the class SPCMtx. The arrays SPCMtx::a1band, SPCMtx::b1band, SPCMtx::c1band, SPCMtx::d1band, and SPCMtx::e1band, are equivalent to the NSI2D arrays a1band, b1band, c1band, d1band, and e1band.

An SPCMtx contains a pointer to an Conn2dNodeOrg as a data member.

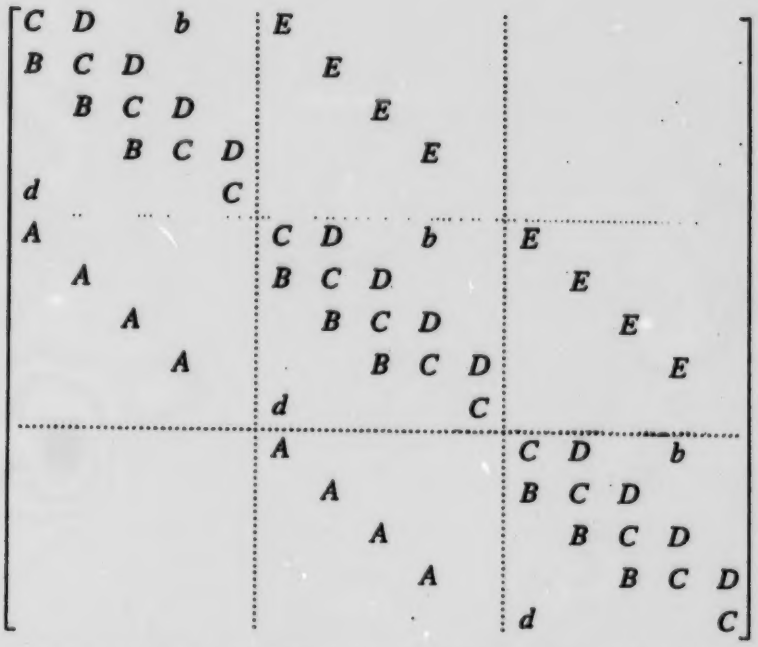


Figure 11: *The matrix corresponding to the O-block of Figure 9 when an implicit wake cut is used.*

The boundary pointers in this node organizer are used by the SPCMtx to determine the locations of the non-zero elements of the matrix which lie off the five principal diagonals.

The SPCMtx's used by the SPC solvers are stocked by NSI2D Fortran subroutines which require direct access to the five diagonals. Hence, no member functions have been supplied for indexing the matrix elements. The SPCMtx constructors and member functions are:

SPCMtx(ConnS2dNodeOrg &b);
Creates an SPCMtx suitable for use on a block with node organizer b.

unsigned block_size() const;
The size of the matrix sub-blocks: either 1, 2, or 3.

unsigned num_rows() const;
unsigned num_cols() const;
Return the dimensions of the matrix.

void get_mem();
Allocates memory for the matrix elements.

void free_mem();
Frees memory for the matrix elements.

void is_mem() const;
True if memory has been allocated for the matrix elements.

In addition, the following function is defined for multiplying vectors (represented by MultiNumVar's with three doubles per node) by the matrix:

```
void matvec(const SPCMtx &m, double alpha, const MultiNumVar &x,
            double beta, MultiNumVar &y);
```

Calculates y = beta*y + alpha*m*x. This function is used by the GMRES solver (see Section 5.3).

5.2 The PreSPCMtx Class

Iterative methods for solving linear systems will converge faster if the matrix is pre-conditioned. The linear system

$$Ax = b \tag{5.1}$$

is converted to

$$(M_1^{-1}AM_2^{-1})(M_2x) = M_1^{-1}b \tag{5.2}$$

where M_1 and M_2 are a non-singular matrices which are chosen so that the product $M_1^{-1}AM_2^{-1}$ approximates the identity matrix. The iterative method will then converge quickly. If M_2 is the identity, then the linear system is said to be left preconditioned by M_1. Conversely, if M_1 is the identity, then the linear system is right preconditioned by M_2.

There are four requirements for a good preconditioner.

1. It should be easy to compute.

2. $M_1^{-1}AM_2^{-1}$ should be a close approximation to the identity matrix so that convergence is fast.

2. It should require little extra memory.

3. The linear systems $M_1x = b$ and $M_2x = b$ should be easy to solve.

In TRANSOM, either left or right ILU0 preconditioning may be used. In this method an incomplete LU-factorization of the matrix is performed. The matrix M (either M_1 or M_2) is represented as a product of three matrices:

$$M = LDU \tag{5.3}$$

where L is equal to the portion of A below the diagonal and is zero elsewhere, D is a diagonal matrix, and U is equal to the portion of A above the diagonal and is zero elsewhere (note that when we speak of the diagonal, each 3×3 block is consider one matrix element). Because of this structure, the linear system $Mx = b$ is easily solved. Moreover, only the diagonal elements need be stored separately. Further details on the ILU0 method, and a more comprehensive discussion of matrix preconditioning, can be found in Reference 19.

Preconditioners for SPCMtx's are represented in TRANSOM by the class PreSPCMtx. It has the following constructors and member functions:

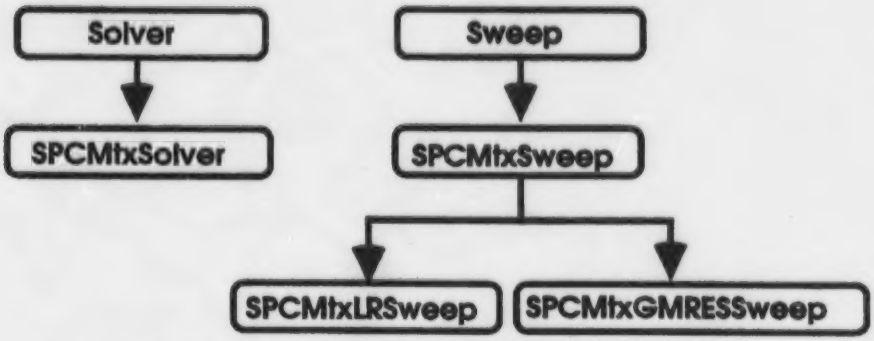


Figure 12: *Class hierarchies of Sweeps and Solvers used by the SPC matrix solver*

PreSPCMtx(SPCMtx &m);
Constructs a preconditioner for the matrix m. A pointer to m is stored so that the matrices L and U do not have to be stored separately.

void make();
Calculates the diagonal matrix D using the current values in the matrix m.

In addition, the following function is defined:

void psolve(const PreSPCMtx &p, MultiNumVar &x, MultiNumVar &b);
Solves the linear system p*x = b for x. The values in the vector b will be changed by this function.

5.3 The SPCMtxSolver Class

The SPCMtxSolver class is a base class for solvers of linear systems having an SPCMtx as their left hand side matrices. It is derived from the TRANSOM Solver class: see Reference 15, Section 4.3. The sweep for an SPCMtxSolver is an SPCMtxSweep. This is an abstract base class which has two fully implemented derived classes, SPCMtxLRSweep and SPCMtxGMRESSweep. The former uses a line relaxation method to solve the linear system and the latter the Generalized Minimum Residual (GMRES) method. The principles of the line relaxation solver are described in Reference 10 while GMRES is described in Reference 19. The inheritance hierarchies for the SPCMtxSolver and SPCMtxSweep classes are shown in Figure 12.

SPCMtxSolver has a single constructor:

SPCMtxSolver(const SPCMtx *m, MultiNumVar *x, const MultiNumVar *b,
PreSPCMtx *p);
Makes a solver for the linear system m*x = b. If p is not null, it will be used as a preconditioner.

The member function solve(), inherited from the base class Solver, is used to solve the system.

Like all classes derived from the base class Solver, an extractor is overloaded for the SPCMtxSolver and OFFSRF_Ifstream classes (see Reference 17, Section 2 for a description of the OFFSRF_Ifstream class). Thus, if slv is a SPCMtxSolver and in_stream is an OFFSRF_Ifstream, then the code

```
in_stream >> slv;
```

will read any of the records described below from the OFFSRF file associated with the stream in_stream. The extractor is used to determine which solution method will be used. To use the line relaxation method use a record of the form

```
{LR METHOD
  {NUMBER OF XI SWEEPS: nxi }
  {NUMBER OF ETA SWEEPS: neta }
  {SWEEP DIRECTION: sweep-dir }
}LR METHOD
```

where *nxi* is the number of relaxation sweeps to perform in the ξ-direction on every execution of the SPCMtxLRSweep, *neta* is the number of relaxation sweeps in the η direction, and *sweep-dir* indicates whether to perform the ξ or η sweeps first: 1 indicates ξ sweeps first, 2 η sweeps first. Since the line relaxation method does not use preconditioning, the preconditioner will be ignored.

To invoke the GMRES method the following input record is used:

```
{GMRES METHOD: preleft restart accuracy }
```

where *preleft* = 1 indicates left preconditioning will be used, *preleft* = 2 indicates right preconditioning, *restart* is the number of GMRES iterations after which the procedure will be restarted, and *accuracy* is the tolerance used to determine when to stop the GMRES iterations. The iterations stop when the norm of the residual has been reduced by the factor *accuracy*.

Given an SPCMtx m, MultiNumVar's b and x, and an OFFSRF_Ifstream in_stream from which to define the method, the linear system m*x = b can be solved using an SPCMtxSolver as follows:

```
SPCMtxSolver slv(m,x,b);    // Construct the solver
in_stream >> slv;           // Determine the solution method by
                            // reading the input file
slv.solve();                // Solve the linear system
```

6 Boundary Conditions

The implementation of boundary conditions is one of the main weaknesses in the current version of TRANSOM. A fully generic method for allowing one set of boundary conditions to be shared among many solvers has not yet been designed. The following sections describe classes used to represent the boundary conditions used by the flow solvers on structured grids.

6.1 The BdyCond Class

The class BdyCond is a base class for boundary conditions in TRANSOM. It has only been partially implemented and so provides a minimal set of services.

The type of boundary condition imposed at a boundary node is denoted by BCType. As currently implemented this is simply an integer. The class BdyCond defines a subset of the possible values of BCType: those boundary conditions which have meaning independent of the solver used.

```
enum BdyCond::BCType { unknown, ignore, solid, symmetric, interior, inflow,
                       outflow, freestream, specify };
```

The condition unknown indicates that no boundary condition has been set. Normally this is used to check for input errors.

The condition ignore is included because some node organizers contain more than one copy of a node. In this case, the boundary condition at a node may be set to ignore. This means that the value at one of the other copies of the node should be used. Some specializations of BdyCond (e.g. SPCBdyCond) require that exactly one copy of a node have a non-ignore condition; this prevents any chance of ambiguity. However, it is possible that the same node will require different boundary conditions in different solvers, so BdyCond does not enforce this rule.

The condition solid indicates that the node is on a solid boundary.

The condition symmetric indicates that the flow is symmetric across the boundary: i.e. the normal component of the velocity and the normal derivatives of scalars such as pressure, k, ε, etc. are zero.

The condition interior indicates that the node is not really on a boundary at all, but is in the interior of the flow. This occurs when the node is on a block boundary which is connected to another block.

The condition inflow indicates that the fluid is flowing into the block across the boundary. Though the BdyCond class does not specify how, advected variables such as velocity, k, or ε, must have their values specified at this node. The method of doing so is left up to individual solvers and/or specializations of BdyCond to handle.

The condition outflow indicates that the fluid is flowing out of the block

across the boundary. Advected variables will not be specified on the boundary but the pressure will be.

The condition freestream is a hybrid of the inflow and outflow conditions. If the current value of the velocity crosses the boundary into the block, then an inflow condition will be used. Otherwise an outflow condition will be used.

The specify condition indicates that values for some of the independent variables will be specified at the current node. How the remaining independent variables are determined is solver-dependent. The manner in which the values are specified is left up to solvers and/or specializations of BdyCond.

The BdyCond class has the following constructors and member functions.

BdyCond(const NodeOrg &norg);
Makes boundary conditions suitable for use on the nodes governed by norg.

BdyCond(const BdyCond &bc);
Creates boundary conditions by copying bc.

virtual ::BCType& operator[](Node n);
virtual ::BCType operator[](Node n) const;
Return the boundary condition type at node n. The return type is ::BCType rather than BdyCond::BCType so that specializations of BdyCond can add different boundary condition types.

int contains(Node n) const;
True if this set of boundary conditions contains a type for node n.

AllNodeIter* get_all_node_iter() const;
Returns an iterator over all the nodes for which this set of boundary conditions has types. See Reference 15, Section 2.2 for a description of the class AllNodeIter.

BdyCond& operator=(const BdyCond &bc);
Copies the boundary condition type at every node in bc which is also in the current set of boundary conditions. This operator provides a generic means of copying boundary conditions from different specializations of BdyCond.

6.2 The SPCBdyCond Class

The SPCBdyCond class is a specialization of BdyCond which represents boundary conditions suitable for the SPC solvers. It has been designed to use data structures compatible with the Fortran data structures used to represent the boundary conditions in NSI2D.

The SPCBdyCond class adds three new boundary condition types to those defined by BdyCond.

enum SPCBCType { trailing_edge, prim_freestream, ri_freestream };

The condition trailing_edge is similar to solid, but only occurs at a node on

a re-entrant boundary. The trailing_edge condition says, in effect, that the solver may use the nodes across the boundary when calculating the values of the pressure. The value of the velocity will be set to zero.

The condition prim_freestream uses implicit boundary conditions in the free stream as described by Rogers and Kwak[10] and as originally implemented in NSI2D. The independent variables are recast as eigenvectors for characteristics for the fluxes normal to the boundary. The eigenvectors for characteristics pointing into the flow region have values set according to the values of the independent variables in the free stream. The eigenvectors for characteristics pointing out of the flow region use first order backward (upstream) differencing.

The condition ri_freestream uses the Riemann invariant free stream boundary condition described in Reference 9, Section 5.1.1.

In contrast, when used in the SPC solvers, the condition freestream checks the direction of flow relative to the boundary and uses an inflow condition if the flow is inward, an outflow condition if it is outward. The inflow condition sets the velocity to its free stream value and extrapolates the value of the pressure from the interior nodes. Conversely, the outflow condition sets the pressure to its free stream value and extrapolates the velocity.

SPCBdyCond stores values for the independent variables in the free stream. These values are used by the inflow, outflow, freestream, prim-freestream, and ri_freestream boundary conditions. By default the free stream has unit magnitude, flows in the positive x direction, and has a pressure of zero. When the k-ε turbulence model is used, the default value for k in the free stream is 10^{-4} times the square of the free stream velocity magnitude and the default value for ε is the cube of the free stream velocity magnitude (note that this is dimensionally incorrect and so assumes a length scale of one unit). When the Spalart-Allmaras turbulence model is used, the default value for the Spalart-Allmaras viscosity (see Section 8.3) is one tenth of the kinematic viscosity ν.

Each instance of SPCBdyCond is associated with a structured block (an instance of ConnS2dBlock). Default boundary conditions are defined according to the topology of the block.

1. On a block with O topology, the boundary etalo is solid and the boundary etahi is given the freestream condition. The boundaries xilo and xihi are both re-entrant and so are given the interior condition.
2. On a block with C topology, the etalo boundary is solid between the trailing edge points and is given the interior condition at the re-entrant nodes in the wake. The etahi, xilo, and xihi boundaries are all given the freestream condition.
3. On all other blocks, the interior condition is used at all re-entrant nodes and the freestream condition at all other nodes.

The default conditions can be over-ridden by specifying new conditions from an input file: see below.

SPCBdyCond uses the classes S2dFStreamData and SPC2dFStreamData to represent values of independent variables to be used by the specify boundary condition.

The S2dFStreamData class stores values for a single independent variable at every node on the block boundaries. It also stores a flag to indicate how that value is to be used. The flag is of the following type:

enum S2dFStreamData::BCFlag { unused, extrap, vals, deriv };

The value unused indicates that the values at this node will not be used. The boundary condition at this node should not be specify.

The value extrap indicates that the value of the independent variable at the boundary node will be calculated by extrapolating the values at interior nodes.

The value vals indicates that the independent variable will be set to the stored value for this node.

The value deriv indicates that the normal derivative of the independent variable will be set to the stored value at this node. Note that the derivatives are with respect to ξ and η. Hence, on the boundary xlo, "normal derivative" means "derivative with respect to η".

SPCBdyCond allows ignore boundary conditions only in the corner points. Moreover, since each corner point is included twice (once for each boundary on which it lies), exactly one of the two copies of the corner node must have an ignore condition. This prevents any possibility of ambiguity in the boundary conditions at the corners.

The class SPC2dFStreamData stores the data needed for specify boundary conditions for each of the independent variables required by the SPC solvers: velocity, pressure, k, ε, and the Spalart-Allmaras viscosity, $\tilde{\nu}$. These data are stored as five separate S2dFStreamData objects.

The classes S2dFStreamData and SPC2dFStreamData will not be described further as most of their member functions are designed for use in generating arguments to Fortran subroutines. As TRANSOM is converted completely to C++, these functions will become superfluous and will be removed. The two classes themselves will also undergo extensive revision and may disappear completely.

The class SPCBdyCond has the following constructors and member functions.

SPCBdyCond(const ConnS2dBlock &b);

Makes boundary conditions suitable for use on the block b. Default boundary conditions will be set according to topology as follows.

With an O topology, the boundary etalo is set to a wall, the boundaries xilo and xihi are set be interior, and the boundary etahi is set to have the extrapolated free stream condition.

With C topology, the boundary etalo is set to a wall between the trailing edge points, and to be interior elsewhere. The boundaries xilo and xihi have Riemann invariant freestream boundary conditions, and the boundary etahi has extrapolated free stream boundary conditions.

With S topology, all boundary points are set to the extrapolated free stream condition unless they are connected to another boundary. In the latter case they are set to be interior.

```
SPCBdyCond(const SPCBdyCond &bc);
```
Creates an new boundary condition by copying bc.

```
virtual BCType& operator[](Node n);
virtual BCType operator[](Node n) const;
```
Return the boundary condition type at node n.

```
void transpose();
```
Transposes the boundary conditions so that boundaries xilo and etalo are swapped and boundaries xihi and etahi are swapped.

```
BCType& get_bcond(Index i, S2dNodeOrg::BndType bnd) const;
```
Returns the boundary condition for the node with index i on boundary bnd.

The following member functions are provided to support the use of Fortran subroutines imported from NSI2D.

```
S2dFStreamData::BCFlag* xi_bc_u(int upper = 0);
S2dFStreamData::BCFlag* xi_bc_v(int upper = 0);
S2dFStreamData::BCFlag* xi_bc_p(int upper = 0);
S2dFStreamData::BCFlag* xi_bc_k(int upper = 0);
S2dFStreamData::BCFlag* xi_bc_e(int upper = 0);
S2dFStreamData::BCFlag* xi_bc_savisc(int upper = 0);
```
Return pointers to arrays of flags for each of the independent variables. The arrays are for the boundary xilo if upper is false, xihi if it is true. The boundaries etalo and etahi may be obtained by transposing the boundary conditions. The arrays are in a format suitable for use in Fortran subroutines imported from NSI2D.

```
double* xi_bc_u_value(int upper = 0);
double* xi_bc_v_value(int upper = 0);
double* xi_bc_p_value(int upper = 0);
double* xi_bc_k_value(int upper = 0);
double* xi_bc_e_value(int upper = 0);
double* xi_bc_savisc_value(int upper = 0);
```
Return pointers to arrays of values for each of the independent variables.

The boundary conditions in an SPCBdyCond can be specified by reading the following records from an OFFSRF input file.

{XILO: *bc-type ilo ihi* }
{XIHI: *bc-type ilo ihi* }
{ETALO: *bc-type ilo ihi* }
{ETAHI: *bc-type ilo ihi* }

The record name indicates the boundary on which conditions are to be specified. The condition *bc-type* is applied to each node on the boundary with index in the range *ilo* to *ihi*. The specify boundary condition requires sub-records to indicate which independent variables are to be specified and what their values are. The sub-record names are simply the names of the independent variable to be specified followed by a list of the values. If the normal derivative is to be specified, the variable name is followed by 'Derivative'. If the first value is 'all' followed by a single value, all the variable values will be set to that value. Any independent variable which is not included will have its values extrapolated from interior nodes. For example:

```
{XILO: specify 0 10
  {Velocity X: 0.0 0.1 0.2 0.3 0.4 0.5 0.4 0.3 0.2 0.1 0.0 }
  {Pressure Derivative: all 0.0 }
}XILO
```

specifies eleven values of the x component of the velocity on boundary xilo. The values of the normal derivative of the pressure are also all set to zero. Because the y component of the velocity is not specified, its values will be extrapolated from interior nodes.

7 The Pressure-Velocity Solver

The pressure-velocity solver solves Equation (2.1) using an iterative method. It is represented by the class SPC2dPVSolver which is a ResidualSolver (see Reference 15, Section 4.6) whose sweep is of class SPC2dPVSweep. The classes SPC2dPVSweep and SPC2dPVSolver are described in the following sections.

A Part-Of hierarchy for the class SPC2dPVSolver is shown in Figure 13. Each of the data members shown will be described in the following sections. The arrows in this figure show the objects to which pointers point. For example, both slv.sw->mslv->rhs and slv.st->residual point to slv.sw->residual indicating that the residual is used by the sweep, the stop test, and the right hand side of the linear system of equations solved by the solver slv.sw->mslv.

7.1 The SPC2dPVSweep Class

The class SPC2dPVSweep is a sweep for a pressure-velocity solver on a structured block: repeated iterations of the sweep lead to a solution of Equation (2.1). It handles all the computational effort for an SPC2dPVSolver. SPC2dPVSweep is derived from the class ResidualSweep (see Reference 15, Section 4.6.2) so that its sweep is split into two functions: calc_residual(), and correct(). In SPC2dPVSweep, calc_residual() both calculates the residual, R_n, and assembles the matrix for the linear system. The function correct() solves the linear system (approximately) using an SPCMtxSolver. SPC2dPVSweep inherits the data members residual and data from ResidualSweep (see Figure 13). The former stores the pressure and velocity residuals at each node, the latter the corrections to the pressure and velocity (ΔD of Equation 2.13).

As in NSI2D, calc_residual() makes use of the fact that the calculation of the fluxes E and E_v is inherently no different from the calculation of F and F_v provided that the data structures which represent the linear system allow transposition of the indices i and j over the two coordinate directions (see Reference 7 for a full discussion). The function calc_residual() is therefore structured as follows.

Initialize the linear system matrix
Calculate the fluxes in the ξ direction
Assemble the ξ fluxes into the linear system
Transpose ξ and η coordinate directions
Calculate the fluxes in the ξ direction
Assemble the ξ fluxes into the linear system
Transpose ξ and η coordinate directions

Due to the coordinate transpositions, the second time the fluxes in the ξ direction are calculated and assembled, it is really the fluxes in the η direction which are being used.

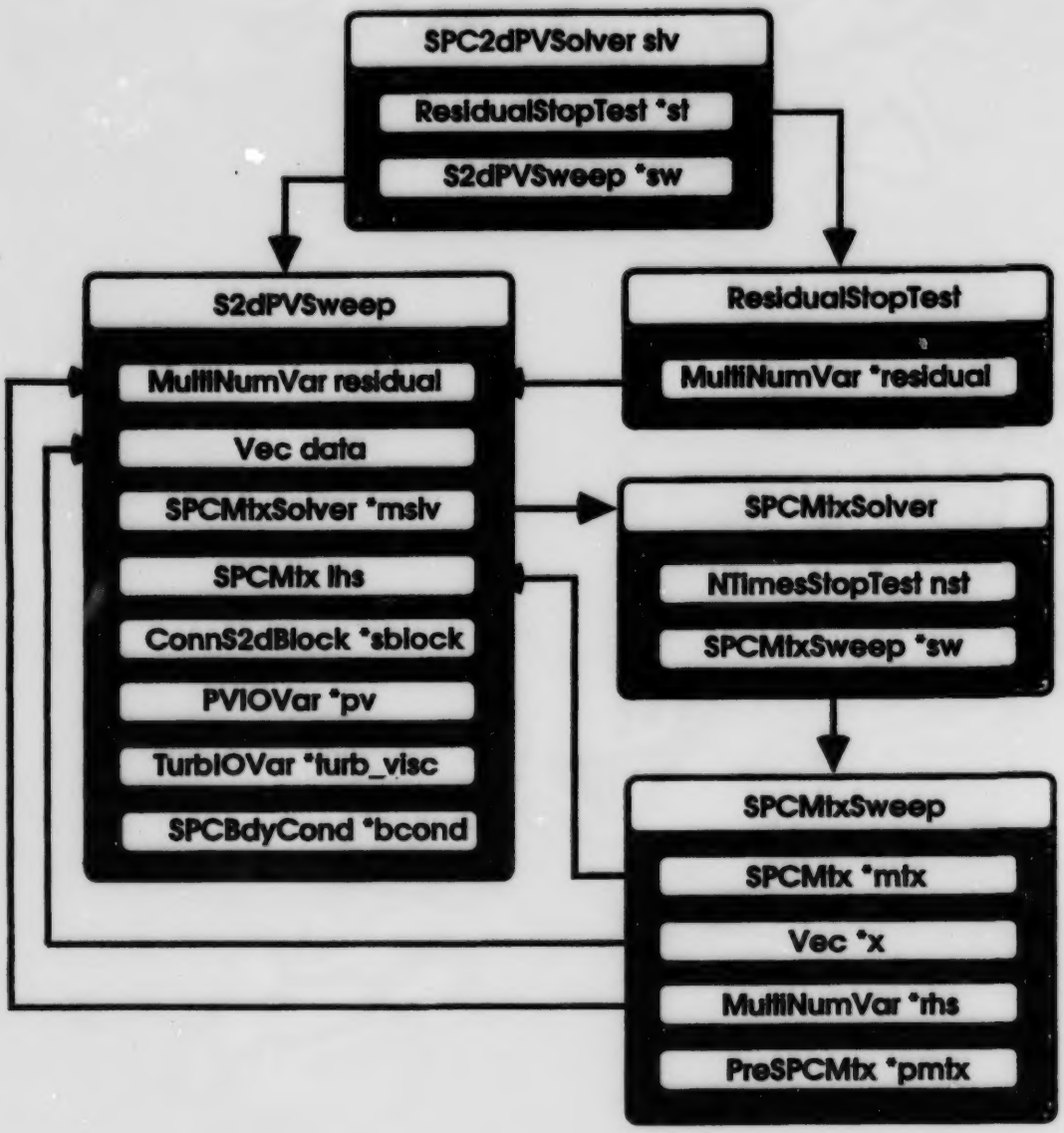


Figure 13: *The SPC2dPVSolver part-of hierarchy.*

The member function correct() simply calculates a preconditioner for the linear system, then calls the SPCMtxSolver function solve() to solve the system.

As shown in Figure 13, an SPC2dPVSweep has eight main data members: the residuals and correction vector inherited from ResidualSweep; a structured block defined as a pointer to a ConnS2dBlock, sblock; boundary conditions, defined by an instance of the class SPCBdyCond, bcond; pressure

and velocity values defined as a pointer to a PVIOVar (see Reference 15, Section 3.5.2), pv; turbulent viscosity defined as a pointer to a TurbIOVar (see Reference 15, Section 3.5.4), turb_visc; an SPCMtx, lhs, to represent the matrix for the linear system; and an SPCMtxSolver, msiv, to solve the linear system. The variables sblock, bcond, pv, and turb_visc are defined from the arguments of the following constructors.

```
SPC2dPVSweep(ConnS2dBlock *b, SPCBdyCond *bc, PVIOVar *pv,
             const TurbIOVar *tv = 0);
```

Constructs an SPC2dPVSweep on block b with pressure and velocity given by pv and turbulent viscosity by tv. If tv is null, laminar flow will be used.

```
SPC2dPVSweep(ConnS2dBlock *b, const IOVar &vars, const Node &n,
             SPCBdyCond *bc = 0);
```

Makes an SPC2dPVSweep on block b, with boundary conditions bc. If bc is null, default boundary conditions will be created according to the block topology (see Section 4.1.4). If the IOVar vars (see Reference 15, Section 3.4 for a description of the IOVar class) contains the fields "Pressure" and "Velocity", then a new PVIOVar will be created which shares these values starting at node n: i.e. the n^{th} value of pressure in vars will be associated with node 0 of block b. Otherwise local pressure and velocity values will be used by creating a PVIOVar whose values are not shared. Similarly, if vars contains the field "Turbulent Viscosity", a new TurbIOVar will be created which shares the values of turbulent viscosity with vars starting at node n. Otherwise the flow will be assumed laminar.

This constructor is a convenient means of creating an SPC2dPVSweep which shares the values of global variables (see Reference 15, Section 4.5).

The SPC2dPVSweep class has the following overloaded functions standard to all sweeps.

```
virtual void initialize();
```

Initializes the SPV2dPVSweep prior to execution of the first sweep. Ensures that the metrics for the block have been calculated and checks boundary conditions for consistency.

```
virtual void sweep();
```

Executes the sweep. Calculates a new approximation for the pressure and velocity using the current values of the turbulent viscosity.

```
virtual void finalize();
```

Writes information to standard output once the iterations have been completed.

An SPC2dPVSweep uses seven constants which can be changed to adjust the solution method.

1. The Reynolds number.

2. The flow angle: This parameter is used to set the initial values of the velocity if they have not already been initialized by reading their values from a file. It is also used to set the velocity values in the free stream for free stream boundary conditions: see Section 6. The default angle is zero.
3. The pseudo-compressibility constant, β: see Equation (2.1). The default value is 50.
4. The boundary dissipation factor, ε: This parameter adjusts the amount of dissipation at boundaries. It should have a value between zero and one. Low values reduce the amount of dissipation but also decrease the stability of the solution. See Reference 6, Section 3.2.1 for more details. The default value is 0.01.
5. The pseudo-time step, Δt: see Equation (2.13). Because of the implicit method of solution, Δt can have very large values without loss of stability. As $\Delta t \to \infty$, the solution method approximates the Newton-Raphson method. The default value is 10^{12}.
6. The flux difference order: This parameter adjusts the order of the finite difference approximation used for the derivatives in the convective terms of Equation (2.2). It may have the values 1, 3, or 5. See Reference 6, Section 3.2.1 for more details. The default value is 5.
7. The implicit wake flag: This logical flag is only of use for blocks with re-entrant boundaries. If true, the linear system is adjusted as described in Reference 11 to ensure the equivalence of the values of independent variables at duplicated nodes on the boundary. If false, independent variables at duplicated nodes are treated as totally independent quantities. Convergence is usually improved when the implicit wake flag is true. The default value is false.

The following member functions may be used to set the values of the seven constants.

```
void set_reynolds_number(double);
void set_flow_angle(double);
void set_beta(double);
void set_epsilon(double);
void set_time_step(double);
void set_fd_order(int);
void set_implicit_wake(int);
```

Like all classes derived from the base class Sweep, an extractor is overloaded for the SPC2dPVSweep and OFFSRF_Ifstream classes (see Reference 17, Section 2 for a description of the OFFSRF_Ifstream class). Thus, if sw is a SPC2dPVSweep and in_stream is an OFFSRF_Ifstream, then the code

```
in_stream >> sw;
```

will read any of the records described below from the OFFSRF file associated with the stream in_stream.

The SPC2dPVSweep extractor reads any of the records used to define an SPCBdyCond (see Section 6.2), any of the records used to define an SPCMtxSolver (see Section 5.3), and "Pressure" and "Velocity" records to define the initial values of the independent variables. The "Pressure" and "Velocity" records have the format of IOVar input records: see Reference 15, Section 3.4. In addition, the values of the seven constants may be set using the following record.

{REYNOLDS NUMBER: *reyno* }
{FLOW ANGLE: *angle* }
{PSEUDO-COMPRESSIBILITY CONSTANT: *beta* }
{DISSIPATION FACTOR: *epsilon* }
{PSEUDO-TIME STEP: *time-step* }
{FLUX DIFFERENCE ORDER: *fd-order* }
{IMPLICIT WAKE CUT}

The flow angle is given in degrees. If the record IMPLICIT WAKE CUT is present, the implicit wake flag will be set to true; otherwise it will be false. All of the records are optional.

7.2 The SPC2dPVSolver Class

The class SPC2dPVSolver is very simple, all of its functionality being inherited from its sweep and from its base class, ResidualSolver. Like all Solver's, it has two principal data members: a pointer to a Sweep, sw, which in this case is an SPC2dPVSweep; and a pointer to a StopTest, st, which in this case is a ResidualStopTest. Figure 13 shows the full Part-Of hierarchy for the SPC2dPVSolver class. Note that the base class ResidualSweep ensures that the residuals used by the sweep are the same as those used by the stop test.

The SPC2dPVSolver class has two constructors whose arguments are simply used to construct the sweep.

```
SPC2dPVSolver(ConnS2dBlock *b, SPCBdyCond *bc, PVIOVar *pv,
              const TurbIOVar *tv = 0);
SPC2dPVSolver(ConnS2dBlock *b, const IOVar &vars, const Node &n,
              SPCBdyCond *bc = 0);
```

The SPC2dPVSolver class inherits the extractor overloaded for the Solver and OFFSRF_ifstream classes. This simply calls the extractors defined for each of the sweep and stop test. Thus, if slv is a SPC2dPVSolver and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> slv;
```

will read any of the records understood by the SPC2dPVSweep class (see the previous section) and any of the records understood by the class ResidualStopTest (see Reference 15, Section 4.6.1). The extractor also recognizes the record OUTPUT FILE for specifying the file into which the pressure and velocity are to be written once the solver has completed its iterations. Following is an example of a TRANSOM input file which creates a pressure-velocity solver on an O-block. The contents of the record PV SOLVER are

read by the SPC2dPVSolver extractor.

```
{GRID: 2 6000
  {O BLOCK: 100 60
    {NAME: Example Block }
    {INCLUDE: coords } | Reads coordinates from the file coords
  }O BLOCK
}GRID

{PV SOLVER: "Example Block"
  {FLOW ANGLE: 30 }
  {REYNOLDS NUMBER: 100 }
  {MAXIMUM NUMBER OF ITERATIONS: 100 }
  {OUTPUT FILE: output }
}PV SOLVER
```

Here default boundary conditions are used. Since the block has O topology, the boundary $\eta = 0$ will be solid and the boundary $\eta = 59$ will use a freestream boundary condition. The boundaries $\xi = 0$ and $\xi = 99$ are both re-entrant and so they have interior boundary conditions. The flow angle is 30° and this value will be used to initialize the velocity. The pressure will be initialized to zero. The iteration will continue until the pressure and velocity residuals are smaller than 10^{-4} (the default) or until 100 iterations have been performed, whichever comes first. Values of the pressure and velocity will then be written to the file "output".

8 Turbulence Solvers

TRANSOM includes four different solvers for the turbulent viscosity on structured blocks. The simplest, LaminarSolver, simply sets the turbulent viscosity to zero at all nodes in a block. The remaining three implement different turbulence models: S2dBLSolver use the Baldwin-Lomax turbulence model[12]; S2dSASolver uses the Spalart-Allmaras turbulence model[13]; and S2dCPSolver uses the Chen-Patel variant of the k-ε model[14].

Because the same turbulence model can be used by many different solvers, TRANSOM has classes to represent the turbulence models themselves as well as classes which solve the turbulence models. The turbulence model classes encapsulate the features of each turbulence model independent of the way it is solved. Thus, they can be used by both structured turbulence solvers and FE turbulence solvers. Base classes for turbulence models are described in the next section; classes used to represent the Baldwin-Lomax, Spalart-Allmaras, and Chen-Patel models are described in Sections 8.2, 8.3, and 8.4, respectively.

8.1 Turbulence Models

The turbulence model solvers have sweeps which use the turbulence model classes as mix-in base classes. The general hierarchy is shown in Figure 14. Each turbulence model class inherits from the base class TurbModel. Each turbulence solver sweep inherits both from the base class Sweep and from a specialization of TurbModel. Thus, the S2dBLSweep inherits both from Sweep and BLTurbModel. Because the Spalart-Allmaras and Chen-Patel models share many data structures, they are both derived from the common base class S2dTransTurbModel (Structured 2-dimensional Transport Turbulence Model). These classes are described in more detail in the following sections.

8.1.1 Turbulence Regions

In TRANSOM, whether the flow is laminar or turbulent is a property which can be specified node by node. The class TurbRegion is used to represent this property.

8.1.1.1 The TurbRegion Class

The class TurbRegion is a general, node by node, representation of the level of turbulence. The type TurbType is used for the level of turbulence at each node. There are three levels of turbulence: laminar, transitional, and turbulent.

```
enum TurbType { lam_flow = 1, trans_flow = 0, turb_flow = 2 };
```

The default level at each node is turbulent.

The following constructors and member functions are defined for the TurbRegion class.

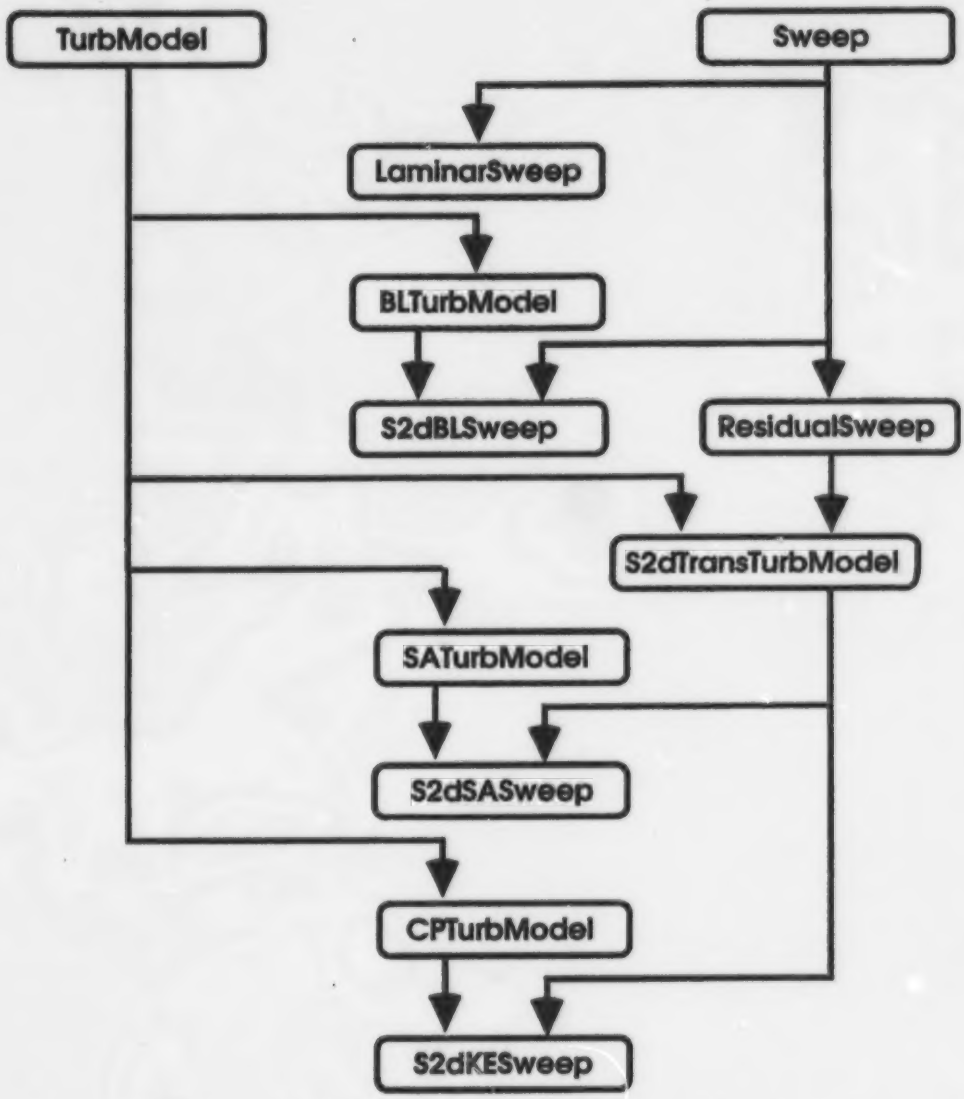


Figure 14: *Class hierarchy for the turbulence models and sweeps .*

TurbRegion(const NodeOrg &norg);
Creates a TurbRegion on the set of nodes governed by the node organizer norg.

void set_all(TurbType t);
Sets the turbulence level at all nodes to t.

virtual void make_turbulence_map();
Sets default turbulence levels for all nodes. In the base class, this function sets the level at each node to turbulent.

double level(Node n) const;

Returns the level of turbulence at node n. A value of 0.0 indicates laminar flow, 0.5 transitional flow, and 1.0 turbulent flow. The return value is a double rather than a TurbType to allow for the possibility of more gradual transition regions.

A turbulence region can be defined in an OFFSRF input file using the following records:

{TURBULENT NODES: $node_1 \cdots node_n$ }

{LAMINAR NODES: $node_1 \cdots node_n$ }

{TRANSITION NODES: $node_1 \cdots node_n$ }

8.1.1.2 The STurbRegion Class

The S2dTurbRegion class is a specialization of TurbRegion which defines a turbulence region on a structured block. Its constructor has the prototype:

S2dTurbRegion(const S2dNodeOrg &norg);

Creates a S2dTurbRegion on the set of nodes governed by the node organizer norg.

The structured block allows more efficient means of assigning turbulence levels to regions of nodes. Two new methods are defined. First, one can assigned a turbulence level to any rectangular region of nodes within the block:

void set_level(TurbRegion::TurbType t, XiIndex ilo, XiIndex ihi, EtaIndex jlo, EtaIndex jhi);

Sets turbulence level to t in the rectangular region of nodes $\{(\xi,\eta): ilo \le \xi \le ihi; jlo \le \eta \le jhi\}$

This may also be done using the following records in an OFFSRF input file:

{TURBULENT REGION

ξ_{lo1} ξ_{hi1} η_{lo1} η_{hi1}

ξ_{lo2} ξ_{hi2} η_{lo2} η_{hi2}

$\vdots$ $\vdots$ $\vdots$ $\vdots$

ξ_{lon} ξ_{hin} η_{lon} η_{hin}

}TURBULENT REGION

{TRANSITION REGION

ξ_{lo1} ξ_{hi1} η_{lo1} η_{hi1}

ξ_{lo2} ξ_{hi2} η_{lo2} η_{hi2}

$\vdots$ $\vdots$ $\vdots$ $\vdots$

ξ_{lon} ξ_{hin} η_{lon} η_{hin}

}TRANSITION REGION

{LAMINAR REGION
ξ_{lo1} ξ_{hi1} η_{lo1} η_{hi1}
ξ_{lo2} ξ_{hi2} η_{lo2} η_{hi2}
⋮ ⋮ ⋮ ⋮
ξ_{lon} ξ_{hin} η_{lon} η_{hin}
}LAMINAR REGION

Each set of four integers defines a rectangular sub-region at which the appropriate turbulence level is set.

The second method is by means of two transition points, ξ_{lo} and ξ_{hi}, on the boundary $\eta = 0$. For nodes with $\xi_{lo} \le \xi \le \xi_{hi}$, the turbulence level is set to laminar. For nodes with $\xi = \xi_{lo}$ or $\xi = \xi_{hi}$, the turbulence level is set to transitional. Otherwise, the turbulence level is set to turbulent. This behaviour is what is usually desired when O and C topology blocks are used. It is a hold-over from NSI2D.

The transition points are implemented using the following member functions:

void set_transition_points(XiIndex ilo, XiIndex ihi);
Sets the surface transition points and adjusts the turbulence levels at all nodes.

The transition points can also be set using the following record in an OFFSRF input file:

{SURFACE TRANSITION POINTS: *itrlo itrhi* }

where *itrlo* and *itrhi* are the ξ values of the two transition points.

8.1.2 The WallDistIOVar Class

Many turbulence models require the distance to the closest wall as a parameter. Since in a multi-block grid it is possible that the closest wall to a given node may lie in a different block, it is best that the wall distance be a global variable. The class WallDistIOVar has been created for this purpose. It is a specialization of the class IOVar (see Reference 15, Section 3.4) which has a single active field called "Wall Distance". It has three constructors which correspond closely with the IOVar constructors.

WallDistIOVar(const NodeOrg &norg, double *d = 0);
Creates a WallDistIOVar for with values at the nodes governed by norg. If d is non-null, the distance values will be stored starting at memory location d; otherwise new memory will be allocated (and destroyed when the WallDistIOVar is destroyed).

WallDistIOVar(const NodeOrg &norg, const IOVar &vars, const Node &n);
Creates a WallDistIOVar which shares the distance values with the IOVar vars starting at node n. If vars does not contain a field named "Wall

Distance", a fatal error will result.

WallDistIOVar(const WallDistIOVar &wd);
A copy constructor; new memory is allocated for the distance values and all values are copied to the newly constructed WallDistIOVar.

The member function dist allows easy access to the distance values; given a Node argument, it returns the distance value at that node. Thus, if w is a WallDistIOVar, then w.dist(n) is the distance of the node n to the nearest wall.

If the wall distance is defined as a global variable in a GLOBAL VARIABLES record, then it will be used by all the structured turbulence solvers. However, currently there is no global method for calculating the wall distance in TRANSOM. If required, each turbulent solver will calculate the wall distance for the nodes in a single block, checking only the nodes on the boundary of that block. Thus, the case in which the closest wall is outside the block will not be handled correctly. Future versions of TRANSOM will include a global method for calculating the wall distance over a whole grid.

8.1.3 The TurbModel Class

The class TurbModel is the base class for all turbulence models. It has the following constructors and member functions:

TurbModel(PointColl *pc, TurbIOVar *tin, const VelIOVar *vin, WallDistIOVar *wd,
TurbRegion *trin = 0);
Creates a TurbModel on the block pc. The turbulent viscosity is stored in tin, the velocity in vin, and the wall distance in wd. The level of turbulence at each node is given by the TurbRegion trin. If trin is null, a new TurbRegion will be allocated and freed when the TurbModel is destroyed.

void set_reynolds_number(double);
Sets the Reynolds number to be used by the turbulence model.

void initialize();
Initializes the model.

A TurbModel can read the following records from an OFFSRF input file to define the Reynolds number and the turbulence levels.

{REYNOLDS NUMBER: reyno }
{TURBULENT NODES: $node_1 \cdots node_n$ }
{LAMINAR NODES: $node_1 \cdots node_n$ }
{TRANSITION NODES: $node_1 \cdots node_n$ }

8.1.4 The S2dTransTurbModel Class

The S2dTransTurbModel class represents a two-dimensional transport turbulence model on a two-dimensional structured grid. It encapsulates the common features of the Spalart-Allmaras and Chen-Patel models when used

on structured grids.

The S2dTransTurbModel class is derived from TurbModel so that it inherits the basic features of all turbulence models (velocity values at the nodes and a TurbRegion to define regions of laminar and turbulent flow).

The turbulent transport equations can be written in the form of Equation (2.9) and discretized in the form of Equation (2.13). Accordingly, the S2dTransTurbModel class is also derived from the class ResidualSweep (see Reference 15, Section 4.6.2). It also stores values for the time step, Δt, as well as an SPCMtx to represent the left hand side of the linear system to be solved at each times step, an SPCMtxSolver to solve the linear system, and an SPCBdyCond to specify the boundary conditions. Figure 14 shows the inheritance hierarchy for all the turbulence models and sweeps.

The S2dTransTurbModel class allows the time step to be scaled by the square root of the Jacobian at each node (see References 8 and 20). Local time stepping can enhance the convergence rate of the k-ε equations considerably.

The S2dTransTurbModel class has the following constructors and member functions:

```
S2dTransTurbModel(ConnS2dBlock *b, SPCBdyCond *bc, IOVar *ivars,
                  TurbIOVar *tin, const VelIOVar *vin, WallDistIOVar *wd,
                  S2dTurbRegion *trin = 0);
```

Creates an S2dTransTurbModel on block b with boundary conditions specified by bc. The independent variables are given by ivars; for the Chen-Patel model ivars will contain k and ε, while for the Spalart-Allmaras model it contains the Spalart-Allmaras viscosity, $\tilde{\nu}$. The turbulent viscosity is in turb_visc, values of the velocity at each node in vel, and the distance of each node to the nearest wall in wd. A turbulence region may be given using the argument trin; if null, a turbulence region will be created.

```
void initialize();
```

Initializes the S2dTransTurbModel prior to the first sweep. Ensures that the metric functions have been calculated, that the distance to the wall has been initialized, and calculates the values of the scaled time step at each node.

```
void set_time_step(double);
```

Set the value of the time step Δt.

An S2dTransTurbModel can read any of the OFFSRF records which can be read by the classes TurbModel, ResidualSweep, SPCMtxSolver, TurbRegion, or SPCBdyCond. In addition it can read the following record to set the type of time stepping to global (time step the same at every node) or local (time step scaled by the square root of the Jacobian).

```
{TIME STEP TYPE: local }
```

or

{TIME STEP TYPE: global }

8.2 The Baldwin-Lomax Solver

The Baldwin-Lomax turbulence model is a two-layer algebraic model in which the turbulent viscosity is evaluated along profiles normal to the walls and wakes. The turbulent viscosity is given by:

$$\nu_t = \begin{cases} (\nu_t)_{inner} & \text{if } y \le y_c \\ (\nu_t)_{outer} & \text{if } y \le y_c \end{cases} \tag{8.1}$$

where y is the distance to the wall. In the inner region the Prandtl-van Driest formula is used:

$$(\nu_t)_{inner} = \ell^2 |\omega|; \quad \ell = k_b y[1 - \exp(-y_+ / A_+)]; \quad y_+ = \frac{y}{\nu}\sqrt{\frac{\tau_w}{\rho}} \tag{8.2}$$

where ω is the vorticity, τ_w, is the shear stress at the wall, and ρ is the fluid density. In the outer layer,

$$(\nu_t)_{outer} = K_b C_{cp} F_w F_k(y); \quad F_k(y) = \left[1 + 5.5\left(\frac{C_k y}{y_{max}}\right)^6\right]^{-1} \tag{8.3}$$

where

$$F_w = \min(y_{max} F_{max}, C_w y_{max} V_d^2 / F_{max}) \tag{8.4}$$

and F_{max} is the value of the function

$$F(y) = y|\omega|[1 - \exp(-y_+ / A_+)] \tag{8.5}$$

in the profile. The quantity y_{max} is the value at which F_{max} occurs and V_d is the difference between the velocity magnitude at y_{max} and the minimum velocity magnitude in the profile. In wakes, the exponential term in ℓ and $F(y)$ is set to zero.

The location of the transition between inner and outer wakes, y_c is the smallest value of y for which $(\nu_t)_{inner} = (\nu_t)_{outer}$.

The Baldwin-Lomax model uses six constants. In the standard model these constants have the following values.

$$k_b = 0.4; \; A_+ = 26; \; C_{cp} = 1.6; \; C_k = 0.3; \; C_w = 1.0; \; K_b = 0.0168$$

8.2.1 The BLTurbModel Class

The solver-independent aspects of the Baldwin-Lomax turbulence model[12] are represented by the class BLTurbModel. It is derived from the base class TurbModel and inherits all its member functions and OFFSRF file input records. In TRANSOM it can only be used on structured grids with C-topology. Since the block used by the Baldwin-Lomax model is structured, the turbulence region it uses may also be structured (i.e. it is represented by an

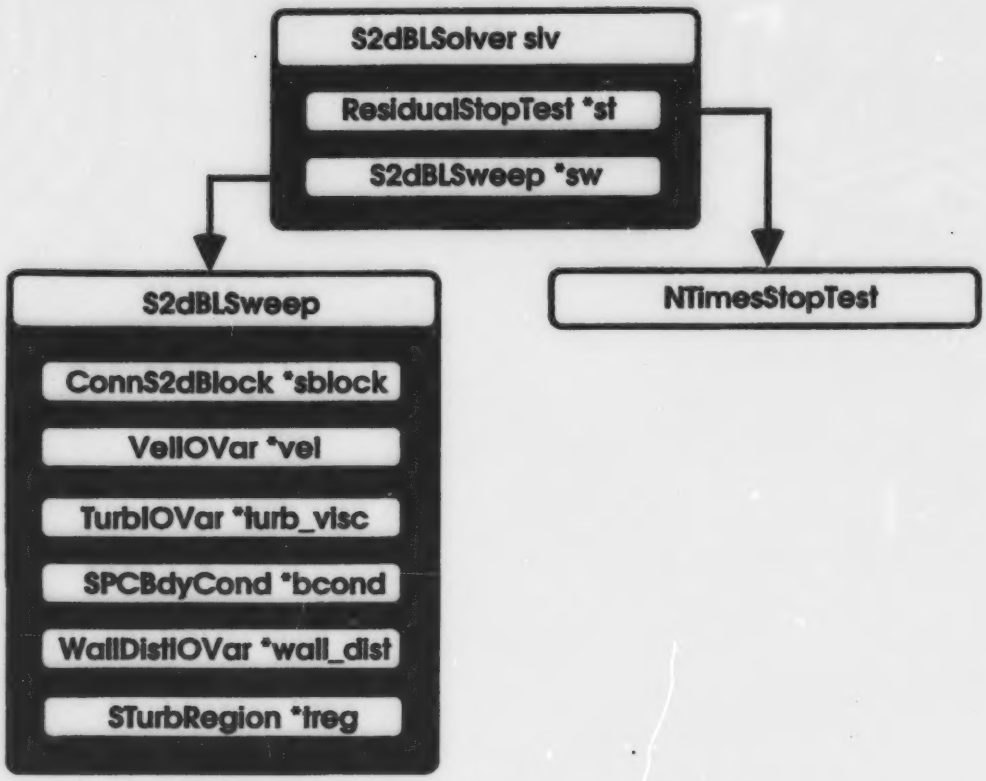


Figure 15: *The S2dBLSolver part-of hierarchy.*

STurbRegion rather than a TurbRegion). Hence, the turbulence region may be defined using the input records LAMINAR REGION, TRANSITION REGION, and TURBULENT REGION (see Section 8.1.1.2).

The six model constants will have the standard values by default but may be changed using the following records in the TRANSOM input file.

{k(b): k_b }
{A+: A_+ }
{C(w): C_w }
{C(cp): C_{cp} }
{C(k): C_k}
{K(b): K_b }

The constructors and member functions of the BLTurbModel class are as follows:

```
BLTurbModel(PointColl *pc, TurbIOVar *turb_visc, const VelIOVar *vel,
            WallDistIOVar *wd);
```

Makes a BLTurbModel on block pc which uses turb_visc to store the

turbulent viscosity. The velocity at each node is given by vel and the distance of each node to the nearest wall by wd.

int is_standard_model() const;
Returns true if the model uses the standard parameters.

8.2.2 The S2dBLSweep Class

The class S2dBLSweep is derived from the classes Sweep (see Reference 15, Section 4.1) and BLTurbModel (see Figure 14). As shown in Figure 15, an S2dBLSweep has six main data members: a structured block defined by an instance of the class ConnS2dBlock; boundary conditions defined by an instance of the class SPCBdyCond; velocity values defined by an instance of the class VelIOVar (see Reference 15, Section 3.5.1); distance of each node to the nearest solid boundary defined by an instance of the class WallDistIOVar; turbulent viscosity defined by an instance of the class TurbIOVar (see Reference 15, Section 3.5.4); and a turbulence region defined by an instance of the class S2dTurbRegion. These variables are all defined using the following constructors.

S2dBLSweep(CTopS2dBlock *b, SPCBdyCond *bc, const VelIOVar *vel,
WallDistIOVar *wd, TurbIOVar *tv, S2dTurbRegion *trin = 0);
Constructs an S2dBLSweep on block b with boundary conditions bc, velocity in vel, wall distance in wd, turbulent viscosity in tv, and turbulence region trin. If any of bc, wd, tv, or trin are null, local variables will be created.

S2dBLSweep(CTopS2dBlock *b, SPCBdyCond *bc, IOVar *vars, const Node &n,
S2dTurbRegion *trin = 0);
Makes an S2dBLSweep on block b, with boundary conditions bc. If bc is null, default boundary conditions will be created according to the block topology (see Section 4.1.4). The IOVar vars (see Reference 15, Section 3.4 for a description of the IOVar class) must contain the field "Velocity". A local VelIOVar will be created which shares the velocity values in vars starting at node n: i.e. the n^{th} value of velocity in vars will be associated with node 0 of block b. If vars contains the fields "Turbulent Viscosity" or "Wall Distance", a new TurbIOVar or WallDistIOVar will be created which shares the values of turbulent viscosity or wall distance with vars starting at node n. Otherwise local unshared variables will be created.

This constructor is a convenient means of creating an S2dBLSweep which shares the values of global variables (see Reference 15, Section 4.5).

The S2dBLSweep class has the following overloaded functions standard to all sweeps.

virtual void initialize();
Initializes the S2dBLSweep prior to execution of the first sweep. Ensures that the metrics for the block have been calculated and

calculates the distance to the solid boundary and wake.

virtual void sweep();
: Executes the sweep. Calculates the turbulent viscosity according to the Baldwin-Lomax model using the current values of the velocity.

virtual void finalize();
: Writes information to standard output once the iteration has been completed.

Like all classes derived from the base class Sweep, an extractor is overloaded for the S2dBLSweep and OFFSRF_ifstream classes (see Reference 17, Section 2 for a description of the OFFSRF_ifstream class). It is simply inherited from the class BLTurbModel. Thus, if sw is a S2dBLSweep and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> sw;
```

will read any of the records described in Section 8.2.1 from the OFFSRF file associated with the stream in_stream.

8.2.3 The S2dBLSolver Class

The class S2dBLSolver solves the Baldwin-Lomax equations on a structured two-dimensional block. Because the Baldwin-Lomax equations are algebraic, no iteration is needed, so the stop test for this solver is chosen to be an NTimesStopTest with count 1: i.e. the sweep will only execute once. The sweep is represented by the class S2dBLSweep.

The class S2dBLSolver is very simple, all of its functionality being inherited from its sweep and from its base class, Solver. It has two constructors whose arguments are simply used to construct the sweep.

```
S2dBLSolver(CTopS2dBlock *b, SPCBdyCond *bc, const VelIOVar *vel,
            WallDistIOVar *wd, TurbIOVar *tv, S2dTurbRegion *trin = 0);
S2dBLSolver(CTopS2dBlock *b, SPCBdyCond *bc, IOVar *vars, const Node &n,
            S2dTurbRegion *trin = 0);
```

S2dBLSolver inherits an extractor from its base class Solver. This recognizes the record OUTPUT FILE for specifying the file into which the turbulent viscosity is to be written, as well as any of the records recognized by the sweep (which in turn uses the extractor defined for the BLTurbModel class). Thus, if slv is a S2dBLSolver and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> slv;
```

will read an OUTPUT FILE record or any of the records understood by the BLTurbModel class (see Section 8.2.1). Following is an example of a TRANSOM input file which creates a Baldwin-Lomax solver on an C-block. The contents of the record BL SOLVER are read by the S2dBLSolver extractor.

```
{GRID: 2 6000
  {C BLOCK: 100 60  25 74
    {NAME: Example Block }
    {INCLUDE: coords } | Reads coordinates from the file coords
```

```
  }C BLOCK
}GRID

{GLOBAL VARIABLES
  Velocity 2
  {Velocity
    u0        v0
    :         :
    u5999     v5999
  }Velocity
}GLOBAL VARIABLES

{BL SOLVER: "Example Block"
  {REYNOLDS NUMBER: 1.0e+06 }
  {SURFACE TRANSITION POINTS: 41 55 }
  {OUTPUT FILE: output }
}BL SOLVER
```

Here default boundary conditions are used. Since the block has C topology, the boundary $\eta = 0$ will be solid between the trailing edge points at $i = 25$ and the boundary $i = 74$. The remainder of this boundary is re-entrant and therefore uses the freestream boundary condition. The freestream boundary condition is used on the remaining boundaries. The velocity is initialized to the values in the Velocity record within the GLOBAL VARIABLES record. The flow will be laminar for nodes whose i indices lie between 41 and 55. A single iteration will be performed and the values of the turbulent viscosity will then be written to the file "output".

It may seem silly to define the Baldwin-Lomax solver as an iterative solver when only one iteration is needed. However, this allows the S2dBLSolver to be included within a concurrent solver (see Reference 15, Section 4.4.1) along with an SPC2dPVSolver with which it shares the velocity. The values of the velocity will then be changed at every iteration and the S2dBLSolver will update the turbulent viscosity appropriately.

8.3 The Spalart-Allmaras Solver

The Spalart-Allmaras turbulence model is a one-equation model in terms of a special viscosity parameter denoted by $\tilde{\nu}$. The turbulent viscosity is given by:

$$\nu_t = \tilde{\nu} f_{v1} \tag{8.6}$$

with

$$f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3} \tag{8.7}$$

and

$$\chi = \frac{\tilde{\nu}}{\nu} \qquad (8.8)$$

where c_{v1} is a constant. The transport of $\tilde{\nu}$ modelled using the following equation:

$$\frac{\partial \tilde{\nu}}{\partial t} + u\frac{\partial \tilde{\nu}}{\partial x} + v\frac{\partial \tilde{\nu}}{\partial y} = \frac{1}{\sigma}\left[\nabla\cdot(\nu+\tilde{\nu})\nabla\tilde{\nu} + c_{b2}(\nabla\tilde{\nu})^2\right] + c_{b1}\tilde{S}\tilde{\nu} - c_{w1}f_w\left(\frac{\tilde{\nu}}{d}\right)^2 \qquad (8.9)$$

with

$$\tilde{S} = |\omega| + \frac{f_{v2}\tilde{\nu}}{\kappa^2 d^2}; \quad f_{v2} = 1 - \frac{\chi}{1+\chi f_{v1}}; \quad c_{w1} = c_{b1}/\kappa^2 + (1+c_{b2})/\sigma; \qquad (8.10)$$

$$f_w = g\left[\frac{1+c_{w3}^6}{g^6+c_{w3}^6}\right]^{1/6}; \quad g = r + c_{w2}(r^6 - r); \quad r = \frac{\tilde{\nu}}{\tilde{S}\kappa^2 d^2} \qquad (8.11)$$

and d is the distance to the closest wall and ω is the vorticity. The seven constants used by the model have the following standard values.

$c_{v1} = 7.1;\ c_{b1} = 0.1355;\ c_{b2} = 0.622;\ c_{w2} = 0.3;\ c_{w3} = 2.0;\ \kappa = 0.41;\ \sigma = 2/3;$

Equation (8.9) is transformed into the (ξ,η) coordinate system in weak conservation form. It is then recast in the form of Equation (2.9) with the residual, R, given by:

$$R = U\frac{\partial \tilde{\nu}}{\partial \xi} + V\frac{\partial \tilde{\nu}}{\partial \eta} - \xi_x\frac{\partial E_1}{\partial \xi} - \eta_x\frac{\partial E_1}{\partial \eta} - \xi_y\frac{\partial F_1}{\partial \xi} - \eta_y\frac{\partial F_1}{\partial \eta} + \tilde{\nu}\left(\xi_x\frac{\partial E_2}{\partial \xi} + \eta_x\frac{\partial E_2}{\partial \eta} + \xi_y\frac{\partial F_2}{\partial \xi} + \eta_y\frac{\partial F_2}{\partial \eta}\right) - S_{SA} \qquad (8.12)$$

with U and V given by Equation (2.5) and

$$E_1 = \left(\frac{\nu + (1+c_{b2})\tilde{\nu}}{\sigma}\right)(\xi_x\tilde{\nu}_\xi + \eta_x\tilde{\nu}_\eta); \quad F_1 = \left(\frac{\nu + (1+c_{b2})\tilde{\nu}}{\sigma}\right)(\xi_y\tilde{\nu}_\xi + \eta_y\tilde{\nu}_\eta) \qquad (8.13)$$

$$E_2 = \frac{c_{b2}}{\sigma}(\xi_x\tilde{\nu}_\xi + \eta_x\tilde{\nu}_\eta); \quad F_2 = \frac{c_{b2}}{\sigma}(\xi_y\tilde{\nu}_\xi + \eta_y\tilde{\nu}_\eta); \quad S_{SA} = c_{b1}\tilde{S}\tilde{\nu} - c_{w1}f_w\left(\frac{\tilde{\nu}}{d}\right)^2 \qquad (8.14)$$

Discretization in time is done as in Equations (2.11) to (2.13), with D replaced by $\tilde{\nu}$. Details of the spatial discretization scheme used to approximate the derivatives in Equation (8.12) may be found in Reference 9, Section 3.2 and Reference 8, Section 3.

8.3.1 The SAViscIOVar Class

The class SAViscIOVar has been defined to store the parameter $\tilde{\nu}$. It is a specialization of the class IOVar (see Reference 15, Section 3.4) which has a single active field called "SA Viscosity". It has three constructors which correspond closely with the IOVar constructors.

SAViscIOVar(const NodeOrg &norg, double *d = 0);

Creates a SAViscIOVar for with values at the nodes governed by norg. If d is non-null, the $\tilde{\nu}$ values will be stored starting at memory location d; otherwise new memory will be allocated (and destroyed when the SAViscIOVar is destroyed).

SAViscIOVar(const NodeOrg &norg, const IOVar &vars, const Node &n);

Creates a SAViscIOVar which shares the $\tilde{\nu}$ values with the IOVar vars starting at node n. If vars does not contain a field named "SA Viscosity", a fatal error will result.

SAViscIOVar(const SAViscIOVar &sa_visc);

A copy constructor; new memory is allocated for the $\tilde{\nu}$ values and all values are copied to the newly constructed SAViscIOVar.

The member function vtilde allows easy access to the distance values; given a Node argument, it returns the $\tilde{\nu}$ value at that node. Thus, if sa_visc is an SAViscIOVar, then sa_visc.vtilde(n) is the value of $\tilde{\nu}$ at node n.

8.3.2 The SATurbModel Class

The solver-independent aspects of the Spalart-Allmaras turbulence model[13] are represented by the class SATurbModel. It is derived from the base class TurbModel and inherits all its member functions and OFFSRF file input records.

The model constants take their standard values by default but may be changed using the following records in the TRANSOM input file.

{KAPPA: κ}
{Cv1: c_{v1}}
{Cb1: c_{b1}}
{Cb2: c_{b2}}
{SIGMA: σ}
{Cw2: c_{w2}}
{Cw3: c_{w3}}
{FREE STREAM SA VISCOSITY: *vtildeinf*}

The value of *vtildeinf* is the ratio of $\tilde{\nu}$ to the kinematic viscosity, ν, in the free stream. Its default value is 0.1.

The constructors and member functions of the SATurbModel class have the following prototypes:

SATurbModel(PointColl *b, SAViscIOVar *sa_visc, TurbIOVar *turb_visc,
const VelIOVar *vel, WallDistIOVar *wd);

Makes an SATurbModel on block b which stores $\tilde{\nu}$ in sa_visc and the turbulent viscosity in turb_visc. Values of the velocity at each node are given by vel and the distance of each node to the nearest wall by wd.

int is_standard_model() const;
Returns true if the model uses the standard parameters.

void initialize();
Sets default values for $\tilde{\nu}$ at each node. If the values in the constructor argument sa_visc have been initialized, then those values are used. Otherwise, if the values in the constructor argument turb_visc have been initialized, then they are converted to values for $\tilde{\nu}$ using the member function tvisc_to_savisc(). If neither have been initialized, the values for $\tilde{\nu}$ are initialized to the free stream values implied by the ratio *vtildeinf*.

double tvisc_to_savisc(double t);
Given a value of the turbulent viscosity, finds the corresponding value of $\tilde{\nu}$.

void tvisc_to_savisc();
At every node in the block, initializes $\tilde{\nu}$ from the values of the turbulent viscosity.

double savisc_to_tvisc(double sa_visc);
Given a value of $\tilde{\nu}$, finds the corresponding value of the turbulent viscosity.

void savisc_to_tvisc();
Initializes the turbulence viscosity from the values of $\tilde{\nu}$.

8.3.3 The S2dSASweep Class

An S2dSASweep is a sweep whose continued application will solve the Spalart-Allmaras transport equations. It is derived from the classes S2dTransTurbModel and SATurbModel (see Figure 14). As shown in Figure 16, an S2dSASweep has eleven main data members: the residual, residual, and correction vector, data, inherited from ResidualSweep; a structured block, sblock, defined as an instance of the class ConnS2dBlock; boundary conditions, bcond, defined as an instance of the class SPCBdyCond; velocity values, vel, defined as an instance of the class VelIOVar (see Reference 15, Section 3.5.1); the distance of each node to the nearest solid boundary, wall_dist, defined as an instance of the class WallDistIOVar; the Spalart-Allmaras viscosity, sa_visc, defined as an instance of the class SAViscIOVar; the turbulent viscosity, turb_visc, defined as an instance of the class TurbIOVar (see Reference 15, Section 3.5.4); a turbulence region, treg, defined as an instance of the class S2dTurbRegion; a linear system represented by a matrix, lhs, of class SPCMtx, and whose right hand side is residual and whose solution vector is data; and a linear system solver, mslv, represented by the class SPCMtxSolver. The residuals, the correction vector, the linear system and its solver are defined internally. The remaining data members all defined using the following constructors.

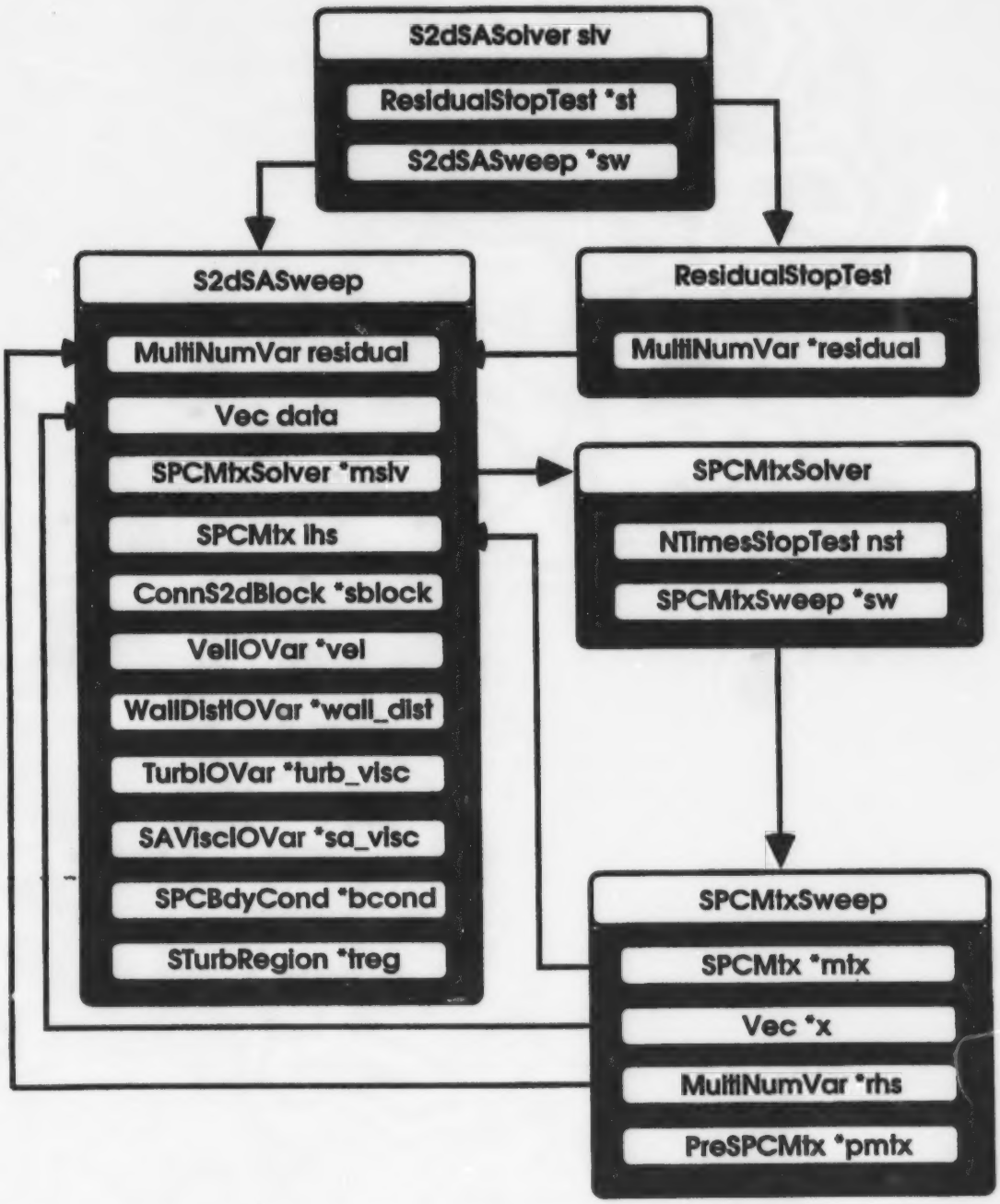


Figure 16: *The S2dSASolver part-of hierarchy.*

S2dSASweep(ConnS2dBlock *b, SPCBdyCond *bc, const VelIOVar *vel,
WallDistIOVar *wd, TurbIOVar *tv, SAViscIOVar *sa_visc,
S2dTurbRegion *tr);

Constructs an S2dSASweep on block b with boundary conditions bc, velocity in vel, wall distance in wd, turbulent viscosity in tv, $\tilde{\nu}$ in sa_visc, and turbulence region trin. If any of bc, wd, tv, sa_visc, or trin are null, local variables will be created.

S2dSASweep(ConnS2dBlock *b, const IOVar &vars, const Node &n,
SPCBdyCond *bc = 0, S2dTurbRegion *trin = 0);

Makes an S2dBLSweep on block b, with boundary conditions bc. If bc is null, default boundary conditions will be created according to the block topology (see Section 4.1.4). The IOVar vars (see Reference 15, Section 3.4 for a description of the IOVar class) must contain the field "Velocity". A local VelIOVar will be created which shares the velocity values in vars starting at node n: i.e. the n^{th} value of velocity in vars will be associated with node 0 of block b. If vars contains the fields "Turbulent Viscosity", "SA Viscosity", or "Wall Distance", a new TurbIOVar, SAViscIOVar, or WallDistIOVar will be created which shares the values of turbulent viscosity, $\tilde{\nu}$, or wall distance with vars starting at node n. Otherwise local unshared variables will be created.

This constructor is a convenient means of creating an S2dSASweep which shares the values of global variables (see Reference 15, Section 4.5).

The S2dSASweep class has the following overloaded functions standard to all sweeps.

virtual void initialize();

Initializes the S2dSASweep prior to execution of the first sweep. Ensures that the metrics for the block and the distance to the walls have been calculated, and checks the consistency of boundary conditions.

virtual void finalize();

Writes information to standard output once the iteration has been completed.

S2dSASweep is derived (via S2dTransTurbModel) from the class ResidualSweep (see Reference 15, Section 4.6.2) from which it inherits the member function sweep(). The sweep is therefore split into two functions: calc_residual(), and correct(). In S2dSASweep, calc_residual() both calculates the residual, R_n, and assembles the matrix for the linear system. The function correct() solves the linear system (approximately) using the SPCMtxSolver.

Like all classes derived from the base class Sweep, an extractor is overloaded for the S2dSASweep and OFFSRF_ifstream classes (see Reference 17, Section 2 for a description of the OFFSRF_ifstream class). It will read any of the records understood by the base classes S2dTransTurbModel and SATurbModel, as well as any of the records understood by the class SPCMtxSolver.

8.3.4 The S2dSASolver Class

The class S2dSASolver solves the Spalart-Allmaras equations on a structured two-dimensional block. It is very simple, all of its functionality being inherited from its sweep and from its base class, ResidualSolver. It has two constructors whose arguments are simply used to construct the sweep.

```
S2dSASolver(ConnS2dBlock *b, SPCBdyCond *bc, const VelIOVar *vel,
            WallDistIOVar *wd, TurbIOVar *tv, SAViscIOVar *sa_visc,
            S2dTurbRegion *tr);
S2dSASolver(ConnS2dBlock *b, const IOVar &vars, const Node &n,
            SPCBdyCond *bc = 0, S2dTurbRegion *trin = 0);
```

The S2dSASolver class inherits the extractor overloaded for the Solver and OFFSRF_Ifstream classes. This simply calls the extractors defined for each of the sweep and stop test. Thus, if slv is a S2dSASolver and in_stream is an OFFSRF_Ifstream, then the code

```
in_stream >> slv;
```

will read any of the records understood by the S2dSASweep class (see the previous section) and any of the records understood by the class ResidualStopTest (see Reference 15, Section 4.6.1). The extractor also recognizes the record OUTPUT FILE for specifying the file into which the Spalart-Allmaras viscosity is to be written once the solver has completed its iterations. Following is an example of a TRANSOM input file which creates a Spalart-Allmaras solver on an O-block. The contents of the record SA SOLVER are read by the S2dSASolver extractor.

```
{GRID: 2 6000
  {O BLOCK: 100 60
    {NAME: Example Block }
    {INCLUDE: coords } ! Reads coordinates from the file coords
  }O BLOCK
}GRID

{GLOBAL VARIABLES
  Velocity 2
  {Velocity
    u0        v0
    :         :
    u5999     v5999
  }Velocity
}GLOBAL VARIABLES

{SA SOLVER: "Example Block"
  {REYNOLDS NUMBER: 1.0e+06 }
  {RESIDUAL ACCURACY: 1.0e-07 }
  {LAMINAR REGION: 32 34 0 59 }
  {LAMINAR REGION: 35 60 0 59 }
  {LAMINAR REGION: 61 63 0 59 }
  {OUTPUT FILE: output }
}SA SOLVER
```

Here default boundary conditions are used. Since the block has O topology, the boundary $\eta = 0$ will be solid and the boundary $\eta = 59$ will use a freestream boundary condition. The boundaries $\xi = 0$ and $\xi = 99$ are both re-entrant and so they have interior boundary conditions. The velocity is initialized to the values in the Velocity record within the GLOBAL VARIABLES record. The flow will be laminar at nodes whose i indices lie between 35 and 60. There are transition regions defined at nodes whose i indices lie between 32 and 34 or between 61 and 63. The flow is turbulent at all other nodes. The iteration will continue until the residuals is smaller than 10^{-7}. Values of $\tilde{\nu}$ will then be written to the file "output".

8.4 The Chen-Patel Solver

In the k-ε model proposed by Launder and Spalding[21], the turbulent viscosity is given by:

$$\nu_t = c_\mu \frac{k^2}{\varepsilon} \tag{8.15}$$

where c_μ is a constant, k is the turbulent kinetic energy, and ε is the rate of dissipation of turbulent kinetic energy. The transport of k and ε is modelled using the following equations:

$$\frac{\partial k}{\partial t} + u\frac{\partial k}{\partial x} + v\frac{\partial k}{\partial y} = \frac{\partial}{\partial x}\left(a_k \frac{\partial k}{\partial x}\right) + \frac{\partial}{\partial y}\left(a_k \frac{\partial k}{\partial y}\right) + \nu_t \Phi - \varepsilon \tag{8.16}$$

$$\frac{\partial \varepsilon}{\partial t} + u\frac{\partial \varepsilon}{\partial x} + v\frac{\partial \varepsilon}{\partial y} = \frac{\partial}{\partial x}\left(a_\varepsilon \frac{\partial \varepsilon}{\partial x}\right) + \frac{\partial}{\partial y}\left(a_\varepsilon \frac{\partial \varepsilon}{\partial y}\right) + c_1 \nu_t \frac{\varepsilon}{k}\Phi - c_2 \frac{\varepsilon^2}{k} \tag{8.17}$$

with

$$a_k = \nu + \frac{\nu_t}{\sigma_k}; \quad a_\varepsilon = \nu + \frac{\nu_t}{\sigma_\varepsilon}; \quad \Phi = 2\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)^2 + 2\left(\frac{\partial v}{\partial y}\right)^2 \tag{8.18}$$

In the Chen-Patel model, the k-ε model is used away from solid boundaries. Close to the boundaries the one-equation Wolfshtein model[22] is used. In this model the expression for the turbulent viscosity includes a damping function f_μ:

$$\nu_t = f_\mu c_\mu \frac{k^2}{\varepsilon} \tag{8.19}$$

with

$$f_\mu = f_\varepsilon^{-1}\left[1 - \exp\left(\frac{-k^{1/2} d}{A_\mu \nu}\right)\right] \tag{8.20}$$

and

$$f_\varepsilon = 1 - \exp\left(\frac{-c_\mu^{3/4} k^{1/2} d}{2\kappa\nu}\right) \tag{8.21}$$

where d is the distance to the closest wall.

Equation (8.16) is used to determine k while ε is given by the algebraic equation

$$\varepsilon = \frac{c_\mu^{3/4} k^{3/2}}{f_\varepsilon \kappa d} \tag{8.22}$$

The seven constants used by the model have the following standard values.

$$c_m = 0.09;\ c_1 = 1.44;\ c_2 = 1.92;\ \sigma_k = 1.0;\ \sigma_\varepsilon = 1.3;\ A_\mu = 70;\ \kappa = 0.418$$

At the nodes far from the wall, Equations (8.16) and (8.17) are transformed into the (ξ,η) coordinate system in weak conservation form. They are then recast in the form of Equation (2.9) with the residual, R, given by:

$$R = U\frac{\partial K}{\partial \xi} + V\frac{\partial K}{\partial \eta} - \xi_x\frac{\partial E_{KE}}{\partial \xi} - \eta_x\frac{\partial E_{KE}}{\partial \eta} - \xi_y\frac{\partial F_{KE}}{\partial \xi} - \eta_y\frac{\partial F_{KE}}{\partial \eta} - S_{KE} \tag{8.23}$$

with U and V given by Equation (2.5) and

$$K = \begin{bmatrix} k \\ \varepsilon \end{bmatrix};\quad E_{KE} = \begin{bmatrix} a_k(\xi_x k_\xi + \eta_x k_\eta) \\ a_\varepsilon(\xi_x \varepsilon_\xi + \eta_x \varepsilon_\eta) \end{bmatrix};\quad F_{KE} = \begin{bmatrix} a_k(\xi_y k_\xi + \eta_y k_\eta) \\ a_\varepsilon(\xi_y \varepsilon_\xi + \eta_y \varepsilon_\eta) \end{bmatrix} \tag{8.24}$$

$$S_{KE} = \begin{bmatrix} \nu_t\Phi - \varepsilon \\ (c_1\nu_t\Phi - c_2\varepsilon)\varepsilon / k \end{bmatrix} \tag{8.25}$$

In the near wall region, the k portion of these equations is retained with ε given by Equation (8.22). The ε equation is effectively removed; in fact, to retain the structure of the linear system, it is altered to be equivalent to $\Delta\varepsilon = 0$. Thus, in the near wall region, ε is not changed after the solution of the linear system, but is later updated using Equation (8,22).

Discretization in time is done as in Equations (2.11) to (2.13), with D replaced by $\tilde{\nu}$. Details of the spatial discretization scheme used to approximate the derivatives in Equation (8.23) may be found in Reference 8, Section 3.

8.4.1 The CPTurbModel Class

The solver-independent aspects of the Chen-Patel variant of the k-ε turbulence model[14] are represented by the class CPTurbModel. It is derived from the base class TurbModel and inherits all its member functions and OFFSRF file input records.

The model constants take the standard values by default but may be changed using the following records in the TRANSOM input file.

{C(mu): c_μ }
{C1: c_1 }
{C2: c_2 }
{SIGMA K: σ_k }
{SIGMA EPSILON: σ_ε }
{A(mu): A_μ }
{KAPPA: κ }
{FREE STREAM K: *kinf* }
{FREE STREAM EPSILON: *epsinf* }
{KE NODES: $node_1 \cdots node_n$ }
{CONNECTION NODES: $node_1 \cdots node_n$ }
{WOLFSHTEIN NODES: $node_1 \cdots node_n$ }

The values *kinf* and *epsinf* are the values of k and ε in the free stream. The record KE NODES is used to indicate at which nodes the k-ε model is used. Similarly the record WOLFSHTEIN NODES indicates where the Wolfshtein model will be used. The record CONNECTION NODES lists nodes on the boundaries between regions where k-ε is used and where the Wolfshtein model is used. The S2dCPSolver class has methods for defining these regions which take advantage of the structure of the block and are therefore easier to use.

The CPTurbModel class has the following constructors and member functions.

CPTurbModel(PointColl *b, KEIOVar *ke, TurbIOVar *turb_visc, const VelIOVar *vel, WallDistIOVar *wd);
Makes a CPTurbModel on block b which stores k and ε in ke (the class KEIOVar is described in Reference 15, Section 3.4.5) and the turbulent viscosity in turb_visc. Values of the velocity at each node are given by vel and the distance of each node to the nearest wall by wd.

void calc_fmu_feps();
Calculates the wall functions f_μ and f_ε at every node of the block.

int is_standard_model() const;
Returns true if the model uses the standard values for all constants.

void initialize();
Sets default values for k, ε, f_μ, f_ε, and ν_t.

double ke_to_tvisc(double k, double eps, double fmu);
Calculates the turbulent viscosity given k, ε and f_μ.

void ke_to_tvisc();
Initializes the turbulent viscosity at every node using the values of k and ε.

void tvisc_to_ke(double tvisc, double phi, double &k, double &eps);

Calculates k, and ε given the turbulent viscosity and the value of Φ. In the inner region, a Newton-Raphson search is used to find the value of k which yields the correct value of ν_t. In the outer region, a similar search is used but subject to the constraint that the turbulence production and dissipation are balanced. The value of Φ is needed to calculate the production term. This cannot be calculated by CPTurbModel because, having no way to determine the structure of the block, it cannot evaluate the derivatives of the velocity.

double tvisc_k_to_eps(double tvisc, double k, double fmu);

Calculates ε given ν_t, k, and f_μ.

void tvisc_k_to_eps();

Calculates ε at every node using the values of k and ν_t.

8.4.2 The S2dCPSweep and S2dCPSolver Classes

The class S2dCPSolver solves the Chen-Patel equations on a structured two-dimensional block. Its implementation is very similar to the implementation of the Spalart-Allmaras solver in S2dSASolver and so will not be described in detail. It is derived from the base class ResidualSweep and uses an S2dCPSweep for it sweep. The latter class is derived from the base classes CPTurbModel and S2dTransTurbModel. S2dCPSolver has the following two constructors.

```
S2dCPSolver(ConnS2dBlock *b, SPCBdyCond *bc, const VelIOVar *vel,
            WallDistIOVar *wd, TurbIOVar *tv, KEIOVar *ke,
            S2dTurbRegion *tr);
S2dCPSolver(ConnS2dBlock *b, const IOVar &vars, const Node &n,
            SPCBdyCond *bc = 0, S2dTurbRegion *trin = 0);
```

The class KEIOVar is used to store the values of k and ε. It is described in Reference 15, Section 3.4.5. The constructors for S2dCPSweep are similar.

8.5 Turbulence Solver Sweep Connections

The Baldwin-Lomax solver is very fast and robust. The Spalart-Allmaras solver is slower and less robust, because it must solve the transport equations iteratively. The Chen-Patel method is the slowest and least robust of all. The two transport equation methods will sometimes fail within the first few iterations. To avoid this problem, it is sometimes convenient to initialize the Spalart-Allmaras and Chen-Patel solvers using the Baldwin-Lomax solver. Similarly, the Spalart-Allmaras solver can be used to initialize the Chen-Patel solver.

In TRANSOM such an initialization sequence can be used if the turbulence solvers are executed within a sequential solver (see Reference 15, Section 4.4.2). A sequential solver maintains a list of sub-solvers. It executes the first solver until its stop test returns true, then the second, and so on until

all the sub-solvers have been executed. When a sub-solver finishes, it may be necessary for it to transmit information to the next solver in the sequence. The SweepConnection class handles the transmission of such information between solvers (see Reference 15, Section 4.4.3.1). To implement the sequential solution of the turbulence solvers, specializations of SweepConnection must be defined for passing information between the turbulence solvers. Each specialization must overload the virtual function update() which updates the state of the new sub-solver from the state of the old.

A Turb_SA_Connection is a SweepConnection which initializes a Spalart-Allmaras turbulent sweep from the current state of any TurbModel (recall that all the turbulent sweeps have TurbModel as a base class). It has the following constructor and member function.

Turb_SA_Connection(TurbModel *tm, S2dSASweep *sa);
: Creates a connection between the turbulence model tm and the sweep sa.

virtual void update();
: The overloaded update function. Copies the values of the turbulent viscosity from the turbulence model to the Spalart-Allmaras sweep. The values of $\tilde{\nu}$ are then initialized from the turbulent viscosity using the SATurbModel member function tvisc_to_savisc().

In the TRANSOM input file a Turb_SA_Connection can be created using a TURB MODEL TO SA CONNECTION record as in the following example.

```
{GRID: 2 6000
  {O BLOCK: 100 60
    {NAME: Example Block }
    {INCLUDE: coords } ! Reads coordinates from the file coords
  }O BLOCK
}GRID

{GLOBAL VARIABLES
  {Velocity
      :          :
  }Velocity
}GLOBAL VARIABLES

{REYNOLDS NUMBER: 1.0e+06 }

{SEQUENTIAL SOLVER
  {BL SOLVER: "Example Block"
    {NAME: BL Solver }
    {OUTPUT FILE: BLoutput }
  }BL SOLVER
  {SA SOLVER: "Example Block"
    {NAME: SA Solver }
    {OUTPUT FILE: SAoutput }
  }SA SOLVER
  {TURB MODEL TO SA CONNECTION: "BL Solver" "SA Solver" }
}SEQUENTIAL SOLVER
```

Notice that the solvers must be named so that they can be referred to by the TURB MODEL TO SA CONNECTION record.

A Turb_CP_Connection is a SweepConnection which initializes a Chen-Patel turbulent sweep from the current state of any TurbModel. It is in all respects like the Turb_SA_Connection class except that the sweep is an S2dCPSweep whose k and ε values are initialized using the S2dCPSweep member function keinit(). The corresponding record in the TRANSOM input file is TURB MODEL TO CP CONNECTION.

9 The Full Pseudo-Compressibility Solver

The solvers described in the previous sections, along with the TRANSOM sequential and concurrent solvers (see Reference 15, Section 4.4) provide the means for running flow simulations in many different ways. Normally the pressure-velocity solver will be run concurrently with a turbulence solver, but it is also possible to run the turbulent solvers independently with a fixed velocity field. This can be a very useful exercise when a calculation is failing to converge. Often one can determine whether the source of the difficulty lies with the turbulence solver or with the pressure-velocity solver by running them independently.

Unfortunately, while the used of nested concurrent and sequential solvers is a very powerful and flexible tool, in practice the input files can become quite unwieldy and difficult to understand. Therefore the class SPC2dSolver was designed to provide a simplified interface to the flow solver which represents the "normal mode of operation" for TRANSOM. It mirrors the calculation steps used by NSI2D: a sweep over the pressure and velocity is run concurrently with a sweep over the turbulent viscosity. Therefore the SPC2dSolver class has been implemented as a specialization of the ConcSolver class which represents concurrent solvers. The pressure-velocity solver is an S2dPVSolver, while the turbulence solver is represented by the class S2dTurbSolver.

The turbulence solver, S2dTurbSolver, executes the four turbulence solvers sequentially: the laminar solver followed by the Baldwin-Lomax solver followed by the Spalart-Allmaras solver followed by the Chen-Patel solver. Any subset of these four may be used, but they must be executed in the order given. For example, it is possible to have only the Baldwin-Lomax solver followed by the Spalart-Allmaras solver, but the Chen-Patel solver followed by the Spalart-Allmaras solver is not allowed. The S2dTurbSolver class is implemented as a specialization of the class SeqSolver which represents sequential solvers.

The organization of the sub-solvers within the SPC2dSolver is shown in Figure 1.

9.1 The S2dTurbSolver Class

The S2dTurbSolver class has six principal data members which it shares amongst it sub-solvers: a structured block defined by an instance of the class ConnS2dBlock; boundary conditions defined by an instance of the class SPCBdyCond; velocity values defined by an instance of the class VelIOVar (see Reference 15, Section 3.5.1); distance of each node to the nearest solid boundary defined by an instance of the class WallDistIOVar; turbulent viscosity defined by an instance of the class TurbIOVar (see Reference 15, Section 3.5.4); and a turbulence region defined by an instance of the class S2dTurbRegion. It has the following constructors.

S2dTurbSolver(ConnS2dBlock *b, SPCBdyCond *bc, VelIOVar *vel, TurbIOVar *tv);
Creates an S2dTurbSolver on block b, having boundary conditions bc, velocity vel, and turbulent viscosity tv. If tv is null, new memory for the turbulent viscosity will be allocated. Other turbulence variables (e.g. k, ε, and $\tilde{\nu}$) will be created.

S2dTurbSolver(ConnS2dBlock *b, SPCBdyCond *bc, VelIOVar *vel,
TurbIOVar *tv, const IOVar &vars, const Node &n);
Creates an S2dTurbSolver on block b, having boundary conditions bc, velocity vel, and turbulent viscosity tv. If tv is null, new memory for the turbulent viscosity will be allocated. Other turbulence variables (e.g. k, ε, and $\tilde{\nu}$) will be shared with vars starting at node n if it contains the appropriate fields.

Since S2dTurbModel was designed primarily to ease the creation of input files, its only other significant function is an extractor for OFFSRF_ifstream's. The extractor will read the following records from an OFFSRF file.

1. REYNOLDS NUMBER: Sets the Reynolds number for the solvers.
2. BALDWIN-LOMAX: Creates an S2dBLSolver to be inserted in the sequential solver, then invokes its extractor to read the sub-records. Thus, the BALDWIN-LOMAX record can contain any of the records understood by the S2dBLSolver extractor (see Section 8.2.3).
3. SPALART-ALLMARAS: Creates an S2dSASolver to be inserted in the sequential solver, then invokes its extractor to read the sub-records. Thus, the SPALART-ALLMARAS record can contain any of the records understood by the S2dSASolver extractor (see Section 8.3.4).
4. CHEN-PATEL: Creates an S2dCPSolver to be inserted in the sequential solver, then invokes its extractor to read the sub-records. Thus, the CHEN-PATEL record can contain any of the records understood by the S2dCPSolver extractor (see Section 8.4.2).
5. LAMINAR ITERATIONS: Sets the number of sweeps for which the laminar solver is to be used.
6. BALDWIN-LOMAX ITERATIONS: Sets the number of sweeps for which the Baldwin-Lomax solver is to be used. If a Baldwin-Lomax solver has not already been created by a BALDWIN-LOMAX record, one will be created now.
6. SPALART-ALLMARAS ITERATIONS: Sets the number of sweeps for which the Spalart-Allmaras solver is to be used. If a Spalart-Allmaras solver has not already been created by a SPALART-ALLMARAS record, one will be created now.
7. Any of the records understood by the class STurbRegion: see Section 8.1.1.2. These are used to define the turbulence region data member which is shared among the sub-solvers.

If no CHEN-PATEL record is present, then the Chen-Patel solver will not be used. If neither BALDWIN-LOMAX nor BALDWIN-LOMAX ITERATIONS

record is present, no Baldwin-Lomax solver will be used, and similarly for the Spalart-Allmaras solver.

9.2 The SPC2dSolver Class

The SPC2dSolver class has three principal data members which it shares between its two sub-solvers: a structured block defined by an instance of the class ConnS2dBlock; boundary conditions defined by an instance of the class SPCBdyCond; pressure and velocity values defined by an instance of the class PVIOVar (see Reference 15, Section 3.5.2); and turbulent viscosity defined by an instance of the class TurbIOVar. (see Reference 15, Section 3.5.4). It has the following constructors.

```
SPC2dSolver(ConnS2dBlock *b, SPCBdyCond *bc = 0);
```

Creates an S2dTurbSolver on block b and having boundary conditions bc. If bc is null, new boundary conditions will be created. All independent variables will be created.

```
SPC2dSolver(ConnS2dBlock *b, const IOVar &vars, const Node &n,
            SPCBdyCond *bc);
```

Creates an S2dTurbSolver on block b and having boundary conditions bc. Independent variables will be shared with vars if it contains the appropriate fields; otherwise they will be created.

Like S2dTurbSolver, the only other significant SPC2dSolver function is the extractor. It will read the following records from an OFFSRF file.

1. PV SOLVER: Contains sub-records used to define the pressure-velocity solver. The sub-records are read by the extractor of the pressure-velocity sub-solver and can therefore be any records understood by the class SPC2dPVSolver: see Section 7.2.
2. Any of the records understood by the class S2dTurbModel. These will be read by the extractor of the turbulence sub-solver.
3. Any of the records understood by the class SPCBdyCond. These will be read by the extractor of the boundary condition data member.
4. NUMBER OF PV CONCURRENT ITERATIONS: Used to set the number of times the pressure-velocity solver will be swept before the turbulent viscosity is updated.
5. NUMBER OF TURBULENT CONCURRENT ITERATIONS: Used to set the number of times the turbulence solver will be swept before the pressure and velocity are updated.

In the TRANSOM input file, an SPC2dSolver is created using an SPC SOLVER record. Following is an example of a TRANSOM input file which creates a full SPC solver a C-block.

```
{GRID: 2 6000
  {C BLOCK: 100 60  25 74
    {NAME: Example Block }
    {INCLUDE: coords } ! Reads coordinates from the file coords
  }C BLOCK
```

```
}GRID

{MAXIMUM NUMBER OF ITERATIONS: 100 }

{SPC SOLVER: "Example Block"
  {OUTPUT FILE: output }
  {REYNOLDS NUMBER: 1.0e+06 }
  {SURFACE TRANSITION POINTS: 41 55 }
  {PV SOLVER
    {FLOW ANGLE: 12 }
    {FLUX DIFFERENCE ORDER: 3 }
  }PV SOLVER
  {BALDWIN-LOMAX ITERATIONS: 30 }
  {SPALART-ALLMARAS}
}SPC SOLVER
```

Here default boundary conditions are used. Since the block has C topology, the boundary $\eta = 0$ will be solid between the trailing edge points at $i = 25$ and the boundary $i = 74$. The remainder of this boundary is re-entrant and therefore uses the freestream boundary condition. The freestream boundary condition is used on the remaining boundaries. The flow angle is 12 degrees and will be used to initialize the velocity and the free stream boundary conditions. The pressure is initialized to zero. The flow will be laminar for nodes whose i indices lie between 41 and 55. A maximum of 100 iterations will be performed, but the calculation will stop if the residuals are all less than 10^{-4} (the default). The Baldwin-Lomax model will be used for the first 30 iterations after which the Spalart-Allmaras model will be used. When the calculation is complete, values of the pressure, velocity, and turbulent viscosity will then be written to the file "output".

10 Concluding Remarks

The initial implementation of NSI2D within TRANSOM has been directed by expedience: reducing the coding effort by reusing as much NSI2D Fortran code as possible. This has led to a number of classes whose design has been compromised by the need to conform to existing Fortran code. The design deficiencies described in this section will likely be addressed in future versions of TRANSOM leading to a program which is more logically implemented and hence easier to understand and to modify.

Because the Fortran subroutines require direct access to the representation of some classes, rather than obtaining access via the member functions, these classes are not well encapsulated. When one of these classes must be modified, every Fortran subroutine which uses it must also be modified., a task which can be long and arduous. These problems will diminish as more and more of the Fortran is converted to C++.

A good example of the lack of encapsulation is provided by the SPCMtx class. The Fortran subroutines which stock the matrix use its representation in five diagonal bands. Therefore, it will not be possible to alternative storage schemes without modifying many of the Fortran subroutines. In particular, it will be difficult to create multi-block solvers which use a single global matrix, especially when both SPC and finite element solvers are used.

As noted in Section 6 boundary conditions are not yet well implemented in TRANSOM. Base classes must be defined which are suitable for all types of solvers, whether on structured or unstructured blocks, and which can be defined globally, for use by all the solvers, or locally, so that a solver may use special boundary conditions which apply only to it.

The SPC2dSolver, S2dSASolver, and S2dCPSolver all share similar features by virtue of the fact that they solve a set of transport equations on a structured block. Many of the common features could be combined into a base class inherited by each. This would make it much easier to implement different turbulence models or different methods of coupling pressure and velocity.

In order to be practical, the SPC solvers must be extended so that they can be used on multi-block grids and in three-dimensional flows. Work on multi-block versions of TRANSOM has already begun. A three-dimensional version of the program NSI2D has also been developed under contract[11]. It will be incorporated into TRANSOM in 1997.

References

1. D. Hally, "User's Guide for HLLFLO Version 2.0", DREA Technical Memorandum 93/309, 1993.
2. D. Hally, "Implementation of a Free Surface in Calculations of the Flow into the Propeller Plane of a Ship", in Proceedings of CADMO92: Third International Conference on Computer Aided Design Manufacture and Operation in the Marine and Offshore Industries, Madrid, Spain, October 1992.
3. G. Dhatt and G. Touzot, *Une Présentation de la Méthode des Eléments Finis*, S. A. Maloine, ed., Paris, 1981.
4. F. Pineau, "A Finite Element Based Reynolds Averaged Navier-Stokes Solver for Modelling Three-Dimensional Turbulent Flows," DREA Report 93/106, 1993.
5. D. Hally, "Implementation of a Finite Element Navier-Stokes Solver in TRANSOM", DREA Technical Memorandum TM/97/225, 1997.
6. D. J. Hall and D. W. Zingg and C. R. Ethier, "A Laminar Incompressible Navier-Stokes Flow Solver", DREA Contractor Report CR/93/462, 1993.
7. D. Hally, "Revisions to NSI2D", DREA Technical Memorandum 94/303, 1994.
8. D. J. Hall and D. W. Zingg and C. R. Ethier, "A Two-Dimensional Incompressible Navier-Stokes Turbulent Flow Solver", DREA Contractor Report, CR/94/435, 1994.
9. P. Giansante, D. W. Zingg and C. R. Ethier, "A Two-Dimensional Incompressible Navier-Stokes Turbulent Flow Solver: Code Enhancement and Convergence Acceleration," , DREA Contractor Report, CR/95/490, 1995.
10. S. E. Rogers and D. Kwak, "An Upwind Differencing Scheme for the Incompressible Navier-Stokes Equations," NASA TN 101051, Nov. 1988.
11. M. Blanco, D. W. Zingg and C. R. Ethier, "A Three Dimensional Incompressible Navier-Stokes Solver," , DREA Contractor Report, in preparation.
12. B. S. Baldwin and H. Lomax, "Thin Layer Approximation and Algebraic Model for Separated Turbulent Flows," AIAA Paper 78-257, 1978.
13. P. R. Spalart, S. R. Allmaras, "A One-Equation Turbulence Model for Aerodynamic Flows," AIAA Paper 92-0439, 1992.
14. H. C. Chen and V. C. Patel, "Near-Wall Turbulence Models for Complex Flows Including Separation," AIAA Journal Vol. 26, No. 6, pp. 641-648, June 1988.
15. D. Hally, "TRANSOM: A Multi-Method Reynolds-Averaged Navier-Stokes Solver: Overall Design," DREA Technical Memorandum, in review.

16. D. Hally, "OFFSRF: A System for Representing Ship Offset Data", DREA Technical Memorandum 89/305, 1989.
17. D. Hally, "C++ Classes for Reading and Writing files in OFFSRF Format", DREA Technical Memorandum 94/302, 1994.
18. C. Royal, "TRANSOM: Un solveur multi-bloc et multi-méthode pour les équations de Navier-Stokes," Bassin d'Essais des Carènes Report, Etude 2560, Pièce No. 11, 1997.
19. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donatao, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, "Templates for the Solutions of Linear Systems: Building Blocks for Iterative Methods," available from **templates@cs.utk.edu**.
20. T.H. Pulliam, "Efficient Solution Methods for the Navier-Stokes Equations," Lecture Notes for the von Karman Institute for Fluid Dynamics Lecture Series: Numerical Techniques for Viscous Flow Computations in Turbomachinery Bladings, von Karman Institute for Fluid Dynamics, Brussels, Belgium, Jan. 20-24, 1986.
21. B.E. Launder, D.B. Spalding, "The Numerical Computation of Turbulent Flows," Computer Methods in Applied Mechanics and Engineering, Vol. 3, pp. 269-289, 1974.
22. M. Wolfshtein, "The Velocity and Temperature Distribution in One-Dimensional Flow with Turbulence Augmentation and Pressure Gradient," International Journal of heat and Mass Transfer, Vol. 12, pp. 301-318, 1969.

UNCLASSIFIED
SECURITY CLASSIFICATION OF FORM
(highest classification of Title, Abstract, Keywords)

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence Research Establishment Atlantic
P.O. Box 1012
Dartmouth, N.S. B2Y 3Z7

2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable).

Unclassified

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title).

Implementation of a Pseudo-Compressibility Navier-Stokes Solver in TRANSOM

4. AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

Hally, David

5. DATE OF PUBLICATION (month and year of publication of document)

December 1998

6a. NO OF PAGES (total containing information. Include Annexes, Appendices, etc).

71

6b. NO. OF REFS (total cited in document)

22

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered).

DREA Technical Memorandum

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include the address).

Defence Research Establishment Atlantic
P.O. Box 1012
Dartmouth, N.S. B2Y 3Z7

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant).

1.g.a

9b. CONTRACT NO. (if appropriate, the applicable number under which the document was written).

10a ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document).

DREA TM/98/239

10b OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor).

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(x) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; further distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected).

Unlimited

UNCLASSIFIED
SECURITY CLASSIFICATION OF FORM

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

TRANSOM is a multi-block, multi-method Reynolds-averaged Navier Stokes solver being developed at DREA to address problems associated with the flow around ships and submarines. It is multi-block because the flow is divided into several distinct regions. It is multi-method because a different solution method may be used on each of the flow regions. At present two different methods of solution can be chosen: a finite volume solver based on the pseudo-compressibility method; and a finite element solver which uses the penalty function method to determine the pressure.

TRANSOM is written in C++ following principles of Object Oriented Programming. This document describes the design of the finite volume pseudo-compressibility flow solver in TRANSOM with emphasis on the class hierarchies used to represent structured blocks, structured matrix solvers, and the pseudo-compressibility solver itself. Two companion reports describe the overall design of TRANSOM and the design of the finite element solver.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

Fluid Flow
Turbulence
Reynolds-Averaged Navier-Stokes Equations
Hydrodynamics
Computer Programs
Object-Oriented Design
TRANSOM

UNCLASSIFIED
SECURITY CLASSIFICATION OF FORM